O'REILLY®



图灵程序设计丛书

涵盖SQL Server, PostgreSQL, Oracle, MySQL和DB2

SQL 经典实例

SQL Cookbook

涵盖150多个SQL常见问题及其解决方案

[美] 安东尼·莫利纳罗 著

刘春辉 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

译者简介

刘春辉

程序员，DBA。入行十余年，起先从事企业软件系统研发，后转入互联网行业做数据库运维，专注于数据密集型应用系统的数据库后端设计、实现、运维以及系统改善等工作。对于开源数据库以及中间件有强烈兴趣。

数字版权声明



图灵程序设计丛书

SQL经典实例

SQL Cookbook

[美] 安东尼·莫利纳罗 著

刘春辉 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社

北 京

图书在版编目（CIP）数据

SQL经典实例 / (美) 安东尼·莫利纳罗
(Anthony Molinaro) 著 ; 刘春辉译. -- 北京 : 人民邮
电出版社, 2018.7
（图灵程序设计丛书）
ISBN 978-7-115-48420-8

Ⅰ. ①S… Ⅱ. ①安… ②刘… Ⅲ. ①关系数据库系统
Ⅳ. ①TP311.138

中国版本图书馆CIP数据核字(2018)第099262号

内 容 提 要

本书详细介绍了各种数据库的 SQL 查询技术和一些基础的 SQL 查询语句，并且通过实例操作的方式讲解了如何插入、更新和删除数据等相关知识。另外，本书还介绍了如何使用 SQL 语句进行日期处理，以及一些其他的 SQL 语句查询操作，能够帮助你掌握相关的 SQL 知识。

本书适用于 SQL 开发人员、非 SQL 程序员和 SQL 专家，以及想要学习 SQL 技术的初学者。

◆ 著　　　　[美] 安东尼·莫利纳罗
　译　　　　刘春辉
　责任编辑　朱　巍
　执行编辑　潘明月
　责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　　印刷

◆ 开本：800×1000　1/16
　印张：31.25
　字数：827千字　　　　2018年 7 月第 1 版
　印数：1 – 2 500册　　　2018年 7 月北京第 1 次印刷

著作权合同登记号　图字：01-2016-7594号

定价：149.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

注 1：本书附录 A 和附录 B 纸书不印刷，放在图灵社区该书页面免费供读者下载：www.ituring.com.cn/book/1691。

前言

SQL 是数据库世界的语言。如果你从事与关系数据库相关的开发工作或者数据报表工作，那么把数据存入数据库并将其再次读取出来的能力最终取决于你所具备的 SQL 知识。然而，许多数据库从业人员只是粗浅地使用 SQL，并不了解其强大的数据处理能力。本书致力于改变这种状况，告诉你 SQL 真正能帮你做些什么。

你手里的这本书像一本菜谱，它包含了一系列常见的 SQL 问题及其解决方案。我希望它能对你的日常工作有所帮助。我按照主题组织章节，当你遇到一个不好解决的 SQL 问题时，请查找最相近的章节。浏览一下各节的标题，你很可能会找到解决思路，或至少得到一些启发。

本书汇集了 150 多个实例，篇幅达 500 多页。然而，这样的篇幅仅仅展示了 SQL 实际能力的一个侧面。我们为日常的编程问题找到了不同的 SQL 解决方案，但其数量远未及问题的数量。本书并不打算集齐全部 SQL 编程问题。事实上，那样做徒劳无功。相反，你能在书中找到许多常见问题的解决方案，从中学到的技术将有助于拓宽你的思路并用于解决新问题。



出版社和我本人持续关注那些新的、有价值的 SQL 解决方案。如果你对某个 SQL 问题有聪明的解决办法，不妨分享出来，或许我会将其加入本书的下一版。可以在“联系我们”一节找到我们的联系方式。

为何写作本书

查询、查询，还是查询。我最初的目标是写一本讲解 SQL 查询技术的书，而不是现在的这本涵盖范围如此广泛的 SQL 实例集。一开始，我专注于解释各种 SQL 查询，从相对简单的语句起步，逐步过渡到比较复杂的部分；我盼望你能掌握其中的技巧，进而运用它们解决工作中遇到的实际问题。我希望能把多年职业生涯中积累的许多 SQL 编程技巧传授给你，希望你能从中学到知识，得到启发，最终找到更好的解决方案。我相信在这个过程中你我彼此都会受益良多。从数据库里读取数据看起来简单至极，然而在 IT 的世界里尽可能高效地实现数据检索却是至关重要的事情。关于高效检索的技术应该被广泛地分享和传

播，这样能提升所有人的效率，使大家互帮互助。

想想为数学做出杰出贡献的格奥尔格·康托尔，他最先意识到把一组元素作为整体来研究有着重大意义（即研究集合本身，而不是研究各个构成要素）。最初，康托尔的工作并不为数学界所接受。然而现在，大家不仅接受了它，甚至认为集合论是数学的基础！更重要的是，集合论能有今天的面貌，并非仅仅得益于康托尔一个人的研究工作；通过向同行分享研究成果，其他诸如 Ernst Zermelo、Gottlob Frege、Abraham Fraenkel、Thoralf Skolem、Kurt Gödel 和 John von Neumann 等数学家进一步发展和改进了集合论。分享不仅让大家更好地理解了康托尔的研究工作，也造就了更完善的集合论。

本书的目标

本书的终极目标是带领你看看 SQL 除了解决典型的问题之外，还能做些什么。在这些年里，SQL 获得了长足的发展。过去我们通常使用 C 或者 Java 等过程化编程语言才能解决的问题，现在已经可以直接用 SQL 解决了，而许多开发人员却对此一无所知。本书将带你学习这些方面的 SQL 技术。

不过，为了避免你误解上述文字，我要声明，我真心赞同一句老话："如果东西没坏，就别修理。"举例而言，假定有一个具体的业务问题要解决，你先用 SQL 从数据库里取出了原始数据，然后用其他编程语言写程序，并实现了一些复杂的业务逻辑处理。如果你的代码工作正常并且运行起来性能也不错，那么，保持现状就很好。我不会怂恿你去实现一个"纯 SQL 解决方案"，我只是希望你能意识到，今天的 SQL 不同于 1995 年的 SQL。今天的 SQL 实际上能做更多的事情。

本书的读者对象

本书的独特之处在于，它面向广泛的读者群，且最终呈现出来的内容保持了高品质。我在书中同时提供了复杂的和简单的实例，如果某个问题没有通用的解决办法，我会针对不同的数据库产品提供多种方案供你选择。因此本书的目标读者群确实是广泛的。

- **SQL 初学者**：或许你买了一本教材，想开始学习 SQL；或许你刚开始上第一个学期的数据库必修课，想通过研究实例巩固课堂所学。你可能看到过有人用区区一条查询语句就神奇地把行形式的数据转换成了列形式，或者把某个长字符串拆解成了一组结果集。本书收录的众多实例将解释上述这些神奇查询背后的技术。
- **非 SQL 程序员**：或许你有其他语言的编程经验，而当前的工作急需你掌握别的同事留下的一些复杂的 SQL。本书列出的实例（尤其是后面几章）会把复杂的查询一一分解开来，帮你循序渐进地理解复杂的代码。
- **SQL 开发人员**：对于中级 SQL 开发人员，本书是你梦寐以求的进阶灵丹妙药。（好吧，这话说得太大了。请原谅一位作者对作品的自信。）如果你从很久以前就开始用 SQL 编程了，并且想开始学习窗口函数，那么本书尤其适合你。举例来说，你不再需要把中间计算结果存入临时表；有了窗口函数，你只要一个 SQL 查询就能得出结果。我要再次声明，我并不是在勉强你接受我的观点。但是，如果你还没有适时跟进 SQL 语言最新的变化，请借助本书更新你的技能。

- **SQL 专家**：毫无疑问，你早已经掌握了书中的技巧，甚至已经能够加以变化、灵活运用。那么，本书对你是否还有帮助呢？也许你精通 SQL Server，想要了解 Oracle；也许你只用过 MySQL，又想知道同样的技术在 PostgreSQL 上是如何应用的。本书涉及多个不同的关系数据库管理系统，分别展示了针对不同产品的实例。这是你拓宽知识领域的好机会。

如何使用本书

请一定认真通读前言部分。它包含了一些必要的背景知识及其他信息，这为后续的内容做了适当的铺垫。"平台和版本"一节会告诉你本书涉及哪些关系数据库管理系统。尤其要看一下"本书中用到的表"一节，这样你会熟悉后续章节里反复出现的数据表样例。你也会在"本书使用的约定"一节中看到一些重要的代码和字体风格约定。上述内容都是前言的组成部分。

注意，这是一本 SQL 实例集，它用一系列的代码实例来帮助你解决你可能会遇到的相似或者相同的问题。请不要试图通过本书学习 SQL 语法和基本知识，至少它不适合那些对 SQL 一无所知的读者。本书适合作为补充材料，而不能替代通常的 SQL 教科书。除此之外，下述要点能帮你更好地利用本书。

- 本书中用到了一些数据库厂商提供的函数。如果你不了解这些函数，不妨参考 Jonathan Gennick 的著作 *SQL Pocket Guide*，该书详细解释了这些函数。
- 如果你不曾使用过窗口函数，或者不熟悉 `GROUP BY` 查询，请先阅读附录 A。该附录讲解了 SQL 的分组概念和做法，它也展示了窗口函数的工作原理。窗口函数的引入是 SQL 最为重要的进展之一。
- 请尊重常识！务必了解，本书不可能涵盖你在工作中可能遇到的全部问题。你要做的是把本书提供的实例作为模板或者指南，灵活运用必要的技术来解决你遇到的问题。你可能会这样说："太棒了！这个例子适用于这种特定的数据集，但我遇到的问题与此不同，以至于无法照搬。"在这种情况下，你应该试着找到两者之间的共性。把书中的查询语句拆解开来，先提取出最基本的形式，然后根据需要逐步增加难度。任何查询都从 `SELECT...FROM...` 这种基本形式起步。如果你在它的基础上逐步地添砖加瓦，每次增加一个新的查询项、函数或连接查询，你就不仅能充分理解每个动作会如何影响最终的结果集，也能了解书中的实例与你的实际需求之间有何种差异。最后，你就能修改这些实例，使其适用于你自己的数据集。
- 测试、测试，再测试。本书中反复出现的 `EMP` 表只有 14 行，毫无疑问，你在工作中遇到的任何一张表都可能比它大。因此，我建议用你自己的数据来测试书中的查询，至少要保证它们工作正常。我没有办法了解你的表长什么样，有哪些索引，和其他表存在何种关联。因此，除非你全面地理解了书中的查询技巧以及把它们应用到你的数据里会有什么样的结果，否则请不要盲目地将它们应用于生产环境的代码。
- 大胆尝试，勇于创新。我鼓励你大胆尝试本书未曾提及的那些技巧和做法。虽然我在书中刻意使用了不同数据库的许多函数，但通常而言，还有别的函数也同样适用于解决某个问题。因此，请大胆改写书中提供的代码，演化出你自己的版本。

- 新东西不一定更好。如果你的代码里没有用到新近引入的 SQL 语言特性，并不意味着引入之后它会变得更有效率。在很多情况下，传统的做法可能和新方案一样好，甚至更好。请记住这一点，尤其是在阅读附录 B 的时候。读过本书以后，请不要认为你必须更新或者修改所有的旧代码。你只需要认识到，SQL 相比 20 年前添加了许多新的、非常好的功能，而它们值得你花时间去学习。
- 不要害怕复杂的查询。如果你发现某个查询看起来太复杂，以至于暂时无法理解，不要害怕。当讲解一个问题时，我已经不遗余力地分解每一个查询，从最简单的形式开始逐级变化，直至呈现出完整的解法，我甚至列出了每个中间步骤的执行结果。你可能没办法立刻纵观全局，但只要跟着我的思路走下去，不仅能够理解查询语句是如何被构造出来的，也能看到中间的每一步会得到什么样的结果。最终，你会发现那些复杂的查询并不难理解。
- 在必要的时候进行防御式编程。为了让本书中出现的查询尽量简洁易懂，我去除了代码里的许多防御性措施。以一个计算员工薪酬总和的查询为例。一种可能出现的情况是，表示薪酬的字段被定义成了 `VARCHAR` 类型，以至于存入数据库的可能是混合了数字和字符串的数据。我在书中给出的代码并没有提防这种情况（因此，`SUM` 函数因无法处理字符数据而导致执行失败）。如果你遇到了这样的数据（更准确地说是“这样的问题”），就需要通过额外的代码做一些防范处理，或者把不规范的数据整理好，因为本书中给出的查询并没有考虑这种数字和字符混合出现的情况。我的观点是，略去这类琐碎的细节有助于你聚焦正题，专注理解查询技术。
- 反复练习最重要。掌握查询的最佳办法是亲自动手编程。阅读代码自然大有裨益，但动手练习是更好的做法。你当然要先读懂那些查询并了解其工作原理，但最终只有通过动手实践才能自己写出查询。

注意，本书中的很多例子都是人为设计的。然而，问题本身都来自真实世界，并非人为臆造。我只不过围绕着一小组包含了雇员数据这样的表来构造实例。我尽力帮你先熟悉示例数据，这样你就能把注意力放在每一个实例背后的技术细节上。面对某个具体问题时，你可能会说：“我不需要针对雇员数据做这些查询。”这时请忽略示例数据，聚焦于我为你演示的那些技术。技术是通用的。我和我的同事天天都在用同样的技术解决不同的问题。我们相信你也是这样。

本书不会涉及的内容

由于时间和篇幅限制，一本书无法囊括你可能实际遇到的所有 SQL 问题及其解决办法。以下是本书不会涉及的内容。

- **数据定义**。本书不会涉及诸如创建索引、添加约束、加载数据等 SQL 操作，这一类操作的语法多数会因数据库的不同而呈现出较大差异，因此你最好多参考官方手册。另外，这类任务的难度还没有达到那种需要专门买一本书来寻求解决方案的程度。尽管如此，第 4 章还是提供了一些涉及数据的插入、更新和删除等常见问题的实例。
- **XML**。我一向认为，与 XML 相关的例子不应该出现在 SQL 书里。把 XML 文档存入关系数据库正变得越来越常见，以至于许多关系数据库管理系统都提供了专有的扩展和工具帮助大家获取和处理 XML 数据。处理 XML 通常需要一些过程化的程序代码，因此

不在本书讨论范围之内。XQUERY 等技术完全独立于 SQL，应该会有专门讲解这一类技术的书。

- **SQL 的面向对象扩展**。除非出现更适合处理对象的语言，否则我不赞成在关系数据库里使用面向对象特性和设计。当前一些数据库实现了部分面向对象特性，不过它们更适用于过程化程序设计，而非 SQL 固有的面向集合的问题解决方式。
- **理论层面的争论**。你不会在本书中看到诸如 SQL 是不是关系型编程语言，或者 `Null` 是否应该存在等这一类观点。我把注意力集中在来自真实世界的 SQL 实例上，理论层面的讨论不见于本书。要解决一个问题，你必须对现有的工具善加利用。你只能拥抱现有的一切，而不应该对那些可望不可及的东西念念不忘。



如果你希望学习更多理论知识，Chris Date 的“关系数据库论文集”系列里的任何一本书都会是一个非常好的起点。你也可以去读他的著作《深度探索关系数据库》。

- **数据库优劣之争**。本书提供的实例兼顾 5 种关系数据库管理系统。你自然想知道哪种数据库提供的方案最好或最快。每一家数据库厂商都能给出足够多的资料来证明自己的产品才是最好的，我不想在这里论及此事。
- **数据库标准之争**。许多书都有意回避不同数据库厂商提供的专有函数，本书却热情拥抱这些专有函数。我不会仅仅出于对可移植性的考虑去写一些复杂低效的 SQL 代码。我从来没有见过哪一家公司明令禁止使用专有的扩展。你付钱买了这些特性，为什么不善用它们？

  数据库厂商的专有扩展之所以存在，自有其原因。相较于标准 SQL，专有扩展往往能提供更高的执行效率和更强的代码可读性。如果你喜欢写完全符合 ANSI 标准的代码，那也很好。正如我之前提到的，我并不是要你去将现有代码改个底朝天。如果你的代码严格符合 ANSI 标准并且工作得很好，那也很棒。归根结底，我们都要工作，都要支付账单，并且都想早点下班回家以享受每天的剩余时光。因此，我并不是在暗示纯标准化的做法有问题。让代码跑起来才是最重要的事。但是，我需要声明，如果你在寻找纯标准化的解决方案，那就不应该阅读本书。
- **遗留系统之争**。本书提供的实例用到了写作本书时已经可用的数据库新特性。如果你还在使用某些旧版本的关系数据库管理系统，我提供的解决方案有许多可能都无法适用。技术不会停滞不前，你也不应该墨守成规。如果你需要找到针对旧版关系数据库管理系统的解决方案，可以翻翻多年前的 SQL 书，它们已经提供了足够多的例子。

本书的结构

本书包括 14 章和两个附录。

- 第 1 章介绍非常基本的查询语句。示例包括如何使用 `WHERE` 子句筛选结果集，为结果集里的列取别名，通过内嵌视图实现别名列引用，使用简单的条件逻辑，限制单个查询返回的记录行数，随机返回记录行，以及检索 `Null` 值。大多数示例都非常简单，但部分示例会再次出现在后续较为复杂的实例里。如果你是 SQL 新手或者对这些用法不太熟悉，那么应该好好读一读这一章的内容。

- 第 2 章提供了一些查询结果排序的实例。这一章介绍了 ORDER BY 子句，并将其用于查询结果排序。示例的难度逐步增加，从简单的单列排序到按照子字符串排序，再到按照条件表达式排序。
- 第 3 章通过一些实例来演示如何合并多个表的数据。如果你是 SQL 新手或不熟悉连接查询，我强烈建议你在阅读第 5 章及后续章节之前先读一读这一章的内容。连接查询几乎就是 SQL 的全部内容，要学会 SQL，你必须理解连接查询。这一章的示例包括执行内连接和外连接，识别笛卡儿积，执行基本的集合操作（差集、并集、交集），以及在连接查询中使用聚合函数。
- 第 4 章分别提供了一些关于插入、更新和删除数据的实例。它们中的大多数都很直截了当（甚至可能让你觉得乏味）。然而，有一些操作可能对你非常有用，比如把一个表的若干行插入另一个表，更新数据时使用关联子查询，理解 Null 值的作用，以及掌握多表插入和 MERGE 命令等特性的用法。
- 第 5 章通过一些实例讲解如何获取数据库的元数据信息。找出一个数据库的索引、约束和表往往非常有用。这些简单的例子帮助你获取关于数据库模式的信息。除此之外，这一章也包括一些动态 SQL 示例，例如用 SQL 生成新的 SQL。
- 第 6 章介绍了一些处理字符串的实例。SQL 的字符串解析能力并不出众，但基于数据库的大量专有函数，再加上一点点创意（通常要用到笛卡儿积），你就能完成不少工作。其中一些更加有趣的例子包括计算一个字符在某个字符串里出现过多少次，基于表的若干行生成列表，把列表和字符串转换成行数据，以及从一个字母和数字混合的字符串里提取数值和字符。
- 第 7 章给出了一些常见的数字运算实例。这些例子极具通用性，并且能让你体会到窗口函数在解决涉及动态计算和聚合的问题时有多方便。这一章的示例包括计算累加值；计算平均数、中位数和众数；计算百分比；在聚合运算中排除 Null 值。
- 第 8 章是涉及日期处理的两章中的第 1 章。对于日常任务来说，处理简单的日期运算十分重要。这一章给出的示例包括计算两个日期之间有多少个工作日，以不同的时间单位（天、月、年等）算出两个日期的差值，以及统计一年中有多少个星期一。
- 第 9 章是涉及日期处理的第 2 章。你会发现日常工作中最为常见的日期操作实例，包括返回一年包含的所有天，计算闰年，算出一个月的第一天和最后一天，生成日历，以及填补一个日期范围里缺失的日期。
- 第 10 章通过一些实例演示如何识别指定范围内的值，以及如何创建一系列的值。示例包括自动生成一系列行数据，填补一个数值范围里缺失的值，查找一个范围的开始值和结束值，以及查找连续的值。
- 第 11 章中的例子有时被开发人员忽视，但对于日常开发工作来说至关重要。这些例子绝不比其他例子更难，但我却见到许多开发人员用非常低效的做法来解决同样的问题。这一章的示例包括查找“骑士值”，为结果集分页，跳过表里的某些行，逆序查找，检索靠前的 *n* 行，以及为查询结果排序。
- 第 12 章提供了一些数据仓储和复杂报表生成领域常见的查询。我最初的愿望就是把这一章的内容作为本书的主体部分。这一章的示例包括行列互换（交叉报表），创建数据分组，创建直方图，计算出简单而完整的小计值，在一个动态的行数据窗口之上执行聚合计算，以及基于给定的时间单位做行数据分组。

- 第 13 章介绍了一些与层次化有关的实例。无论采用何种建模方式，你总会在某个时刻需要做数据格式化工作，比如以树形结构或者父子关系形式展现出来。这一章提供的例子能够帮你完成这些任务。利用传统的 SQL 创建树形结构的数据集并不容易，所以数据库厂商提供的专有函数在这一章里显得尤其有用。示例包括呈现数据的父子关系，从根节点到叶子节点逐层遍历，以及构造一个层次结构。
- 第 14 章是各种实例的大杂烩，它们很难被归入某个问题领域，却既有趣又有用。这一章不同于其他各章之处在于，它只聚焦于数据库厂商提供的专有特性。每个实例只针对一种数据库而设，这是全书唯一这么做的一章。有两个原因促使我这么做：第一，我想让这一章的内容既有趣又带些许极客风格；第二，有些实例的存在就是为了突出某个数据库厂商的专有函数，因为在其他关系数据库管理系统里没有等价实现（示例包括 SQL Server 的 `PIVOT`/`UNPIVOT` 操作符和 Oracle 的 `MODEL` 子句）。在某些情况下，你能简单地改造一下这一章提供的解决方案，将其用于另一种数据库。
- 附录 A 带你复习窗口函数的相关知识，并且详细讨论了 SQL 分组查询。你可能不熟悉窗口函数，附录 A 可以帮助你快速入门。此外，根据我的经验，`GROUP BY` 查询的使用一直令许多开发人员感到迷惑。附录 A 精确定义了何为 SQL 分组查询，并且给出了多种查询示例，以进一步解释该定义。接着，附录 A 讨论了 `Null` 值对分组、聚合以及分区的影响，最后讨论了窗口函数中更难理解却功能强大的 `OVER` 子句（即开窗子句）。
- 附录 B 主要是向 David Rozenshtein 致敬，并把我在 SQL 开发方面的成就归功于他。Rozenshtein 的作品 *The Essence of SQL* 是我在课堂之外买的第一本 SQL 书。我当时买这本书，并非为了应付考试。正是这本书教会了我如何用 SQL 思考。时至今日，我仍把自己关于 SQL 工作原理的许多心得体会归功于这本书。与我读过的其他 SQL 书相比，它是如此与众不同，我为它能成为我的第一本 SQL 书而充满感激之情。我在附录 B 中重新审视了 *The Essence of SQL* 里出现过的一些查询语句，并给出了使用窗口函数实现的新解决方案。（在 *The Essence of SQL* 出版时，窗口函数尚未出现。）

平台和版本

SQL 产品日新月异。各个厂商不断地为各自的产品加入新特性和新功能。因此，我要先告诉你本书是为各个数据库产品的哪些版本而准备的。

- DB2 v.8
- Oracle Database 10*g*（除了少数实例，本书的解决方案也都适用于 Oracle8*i* 和 Oracle9*i*）
- PostgreSQL 8
- SQL Server 2005
- MySQL 5

本书中用到的表

本书中的大部分例子都会涉及两个表：`EMP` 表和 `DEPT` 表。`EMP` 表有 14 行数据，它非常简单，仅用到了数字、字符串和日期字段。`DEPT` 表有 4 行数据，它也很简单，只含有数字和字符串字段。这两个表在许多现存的数据库教科书里都曾出现过，大家也早已熟知在员工

和部门之间所存在的多对一关系。

当我谈到示例表的话题时，我想说本书中的示例除了极少数的几个，几乎所有的示例都会用到这两个表。我不会像其他一些书那样，通过修改示例数据来构造一些你绝对不可能在真实世界里实现的解决方案。

说到解决方案，请允许我稍微提一下，只要状况允许，我都会尽可能地为本书涉及的 5 种关系数据库管理系统提供通用的解决方案。但经常无法做到这一点。尽管如此，在许多情况下，多种数据库可能共用一种解决方案。举例而言，因为相互支持对方的窗口函数，所以 Oracle 和 DB2 经常共用解决方案。如果解决方案共用或非常类似，那么在讨论部分也会一并提及。

EMP 表和 DEPT 表的数据分别如下所示。

```
select * from emp;

EMPNO ENAME  JOB        MGR HIREDATE     SAL COMM  DEPTNO
----- ------ --------- ---- ----------- ---- ---- -------
 7369 SMITH  CLERK     7902 17-DEC-1980  800           20
 7499 ALLEN  SALESMAN  7698 20-FEB-1981 1600  300      30
 7521 WARD   SALESMAN  7698 22-FEB-1981 1250  500      30
 7566 JONES  MANAGER   7839 02-APR-1981 2975           20
 7654 MARTIN SALESMAN  7698 28-SEP-1981 1250 1400      30
 7698 BLAKE  MANAGER   7839 01-MAY-1981 2850           30
 7782 CLARK  MANAGER   7839 09-JUN-1981 2450           10
 7788 SCOTT  ANALYST   7566 09-DEC-1982 3000           20
 7839 KING   PRESIDENT      17-NOV-1981 5000           10
 7844 TURNER SALESMAN  7698 08-SEP-1981 1500    0      30
 7876 ADAMS  CLERK     7788 12-JAN-1983 1100           20
 7900 JAMES  CLERK     7698 03-DEC-1981 950            30
 7902 FORD   ANALYST   7566 03-DEC-1981 3000           20
 7934 MILLER CLERK     7782 23-JAN-1982 1300           10


select * from dept;

DEPTNO DNAME          LOC
------ -------------- ---------
    10 ACCOUNTING     NEW YORK
    20 RESEARCH       DALLAS
    30 SALES          CHICAGO
    40 OPERATIONS     BOSTON
```

除此之外，本书还会用到 4 张数据透视表：T1、T10、T100 和 T500。因为这些只是数据透视表，所以我认为不需要给它们取更容易懂的名字。关于表名，跟在字母 T 后面的数字表示该表有几行数据，每行有一个序号，从 1 开始。例如，T1 表和 T10 表的数据如下所示。

```
select id from t1;

        ID
----------
         1
```

```
select id from t10;
        ID
---------
         1
         2
         3
         4
         5
         6
         7
         8
         9
        10
```

顺便说一下，一些数据库支持局部 SELECT 语句。举例来说，可以只有 SELECT 而没有 FROM 子句。我不喜欢这样的用法，因此我构造了 T1 这样只有一行数据的表并针对该表执行查询，而没有使用局部查询。

任何其他仅用于特定实例和章节的表，我会在书中的适当位置做出解释。

本书使用的约定

本书遵循了许多排版和代码编写约定。请花点时间熟悉它们，这有助于加深你对本书内容的理解。代码编写约定尤其重要，因为我不能在每一个实例中都重复强调一遍。因此，我把一些重要的约定都列在下面。

排版约定

本书遵循下列排版约定。

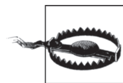
大写字母表示 SQL 关键字。

小写字母用于所有代码示例。诸如 C 和 Java 这样的编程语言都使用小写形式的关键字，我发现其可读性比大写形式更好。因此，本书中所有查询语句都使用小写形式。

等宽粗体用于在交互示例里表示用户输入的内容。



这个图标表示小技巧、建议或者一般性提示。



这个图标表示警告或注意事项。

代码编写约定

我习惯在 SQL 语句中全部用小写字母，不管是关键字还是用户指定的标识符。例如：

```
select empno, ename
  from emp;
```

你的习惯可能不同。例如，许多人喜欢把 SQL 关键字大写。你需要遵循你喜欢的或者项目要求的风格。

尽管代码示例用小写形式，但在正文里 SQL 关键字和标识符始终都是大写字母。我这样做是为了将它们和其他普通文本明确区分开来。例如，前述查询语句展示了一个针对 EMP 表的 SELECT 操作。

尽管本书涵盖了 5 个数据库厂商的产品，但是我还是决定用同样的格式呈现所有产品的输出结果。

```
EMPNO ENAME
----- ------
 7369 SMITH
 7499 ALLEN
…
```

许多实例在 FROM 子句里用到了内嵌视图或子查询。ANSI SQL 标准规定要给它们取别名。（只有 Oracle 不要求指定这一类别名。）因此，我在解决方案里经常用类似 x 和 y 这样的别名来标识内嵌视图。

```
select job, sal
  from (select job, max(sal) sal
          from emp
        group by job) x;
```

注意最后紧挨着圆括号的字母 X。在这里，字母 X 变成了 FROM 子句里那个子查询返回的表的名字。列别名是一个有用的工具，能帮我们写出自注释的代码；相对而言，（本书中出现过的多数）内嵌视图的别名只是一种形式化的东西。通常我会为它们取一个简单的名字，诸如 X、Y、Z、TMP1 和 TMP2。在某些情况下，如果我觉得取一个更好的别名有助于增加可读性，我就会那样做。

你将会看到在每个实例的解决方案部分出现的 SQL 语句，其每一行都会被编号，例如：

```
1 select ename
2     from emp
3  where deptno = 10
```

这些数字并不是语法的一部分，我把它们包含进来只是为了方便在每个实例的讨论部分能使用序号来引用查询语句里的各个部分。

使用示例代码

本书的目的在于帮助你完成工作。一般来说，你可以在你的程序和文档中使用本书的代码。只要不是大规模地复制代码，你就不需要联系我们取得授权。举例而言，你写的一个程序用到了本书的几个代码片段，这是不需要授权的。但是，如果你把书中的示例代码刻录到 CD-ROM，并拿去出售和分发，则需要获得授权。在回答问题时引用本书以及本书的示例代码无须取得授权。但如果要在你的产品文档里收录本书中出现过的大量示例代码，

则需要获得授权。

欢迎你在使用本书的示例代码时注明出处，但这不是强制要求。通常要注明书名、作者、出版社和 ISBN。例如：*SQL Cookbook*, by Anthony Molinaro. Copyright 2006 O'Reilly Media, Inc., 0-596-00976-3。

如果你认为你对示例代码的使用不在合理使用和上述无须授权的范围之内，那么请通过 permissions@oreilly.com 联系我们。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：http://shop.oreilly.com/product/9780596009762.do。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问网站：
http://www.oreilly.com。

我们在 Facebook 的地址是 http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址是 http://www.youtube.com/oreillymedia。

Safari® Books Online



Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM

Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

致谢

如果没有许多人的帮助和支持，就不会有本书。我想感谢我的妈妈 Connie，我把本书献给她。没有她的努力工作和奉献就不会有今天的我。妈妈，感谢您为我做的一切，我很感激您为我和哥哥所做的一切，成为您的儿子是我的荣幸。

我要感谢我的哥哥 Joe。每次我暂停写作，从巴尔的摩回到家休息时，你让我知道与家人相处的时光是多么重要，让我意识到我应该尽快完成写作，以回到生活中更重要的事情中来。你是一个善良的人，我尊敬你。我非常为你自豪，为你是我的哥哥而感到自豪。

我要感谢我出色的未婚妻 Georgia。没有你的支持，我不可能完成这本书。你和我一起见证了本书的创作过程。我知道这对你来说很困难，对我也一样。我白天忙于工作，晚上忙于写作，但你包容了这一切。我永远感激你的理解和支持。谢谢你，我爱你。

我要感谢我未来的岳母 Kiki 和岳父 George。感谢你们在整个过程中对我的支持。无论我去你们家小住，还是去拜访，你们都让我觉得像在自己家里一样，我和 Georgia 每次吃得都很好。我要感谢 Georgia 的姊妹 Anna 和 Kathy。每次回到家和大家一起放松休息的时候总是很开心，谢谢你们提供了一个让我和 Georgia 暂时放下本书的写作从巴尔的摩去和你们聚会的机会。

我要感谢我的编辑 Jonathan Gennick，没有他就不会有本书。Jonathan，你理应因为本书收获排山倒海的赞誉。作为本书的编辑，你的付出超越了一个编辑通常的工作范畴，非常感谢你。从提供素材，到无数次的重写，即使面临即将到来的最后期限也要保持气氛幽默愉快，没有你我不可能完成本书。我很荣幸你是我的编辑，也很感激你给我的这个机会。你既是一名经验丰富的 DBA，又是一位作者，很高兴能和你这样的技术专家合作。我不知道有多少编辑能够放下他们的工作改行做一名专业的 DBA，但我相信 Jonathan 可以。作为一位有 DBA 经验的编辑，你总是理解我要表达什么，即使有时候我不知道如何表达。O'Reilly 有你这样的员工真的很幸运，我也很幸运有你作为我的编辑。

我想感谢 Ales Spetic 和 Jonathan Gennick 的作品 *Transact-SQL Cookbook*。牛顿曾经说过："如果说我看得比前人更远一点的话，那是因为我站在巨人的肩膀上。"在 *Transact-SQL Cookbook* 一书的致谢部分，Ales Spetic 写下的一段话是对这句名言的最好证明，我觉得应该放在每一本 SQL 书里。现在我把这段话也放在本书里。

> 我希望本书能为像 Joe Celko、David Rozenshtein、Anatoly Abramovich、Eugine Berger、Iztik Ben-Gan、Richard Snodgrass 等杰出作者的作品提供一点有益的补充。我花费了无数个夜晚研读他们的作品，我的知识几乎都源于他们的作品。当我写下这些内容的时候，我意识到，每当我花费一个晚上从他们的作品中学到点什么，他们当初必然花费了十个晚上用于将他们的知识凝结成连贯、可读的形式。能够以本书来回馈 SQL 社区是我的荣幸。

我想感谢 *Mastering Oracle SQL* 一书的作者 Sanjay Mishra，感谢他把我介绍给 Jonathan。如果不是 Sanjay，我可能不会认识 Jonathan，也就不可能写作本书了。一封简单的邮件就能改变生活，这多么地不可思议啊。我要感谢 David Rozenshtein，尤其要感谢他写了 *The Essence of SQL*，这本书教会我用集合和 SQL 的方式思考和解决问题。我想感谢 David Rozenshtein、Anatoly Abramovich 和 Eugene Birger，他们的作品 *Optimizing Transact-SQL* 教会了我很多现在仍然经常用到的高级 SQL 技巧。

我想感谢 Wireless Generation 公司的整个团队，这是一家优秀的公司，有着许多优秀的人才。非常感谢所有花时间来检查、讨论、提建议和帮助我完成本书的人：Jesse Davis、Joel Patterson、Philip Zee、Kevin Marshall、Doug Daniels、Otis Gospodnetic、Ken Gunn、John Stewart、Jim Abramson、Adam Mayer、Susan Lau、Alexis Le-Quoc 以及 Paul Feuer。我想感谢 Maggie Ho，她仔细检查了我的书稿，并且针对附录 A 给予了非常有用的反馈。我想感谢 Chuck Van Buren 和 Gillian Gutenberg，他们给了我很有益的关于跑步的建议。每天早上的锻炼让我思路清晰、精神放松。如果不是每天晨练，我想我可能无法完成本书。我想感谢 Steve Kang 和 Chad Levinson，当他们经过白天的辛苦工作，晚上想去 Union Square 的 Heartland Brewery 喝一杯或者吃烧烤的时候，却不得不和我没完没了地讨论各种 SQL 技巧。我想感谢 Aaron Boyd 给予我的支持和善意帮助，最重要的还有他的那些好建议。Aaron 是一个真诚、努力、直爽的人，公司因为有了他而变得更优秀。我想感谢 Olivier Pomel 给予的支持和帮助，尤其是那个根据若干行数据创建列表的 DB2 解决方案。Olivier 贡献了那个解决方案给我，我甚至无须再找个 DB2 系统来做测试！我向他解释了 `WITH` 子句的工作原理，几分钟后他就拿出了你在本书中看到的那个解决方案。

Jonah Harris 和 David Rozenshtein 还帮忙检查了手稿，并且提供了有益的反馈意见。ArunMarathe、Nuno Pinto do Souto 和 Andrew Odewahn 参与了提纲和实例选择的讨论，而在那个时候本书尚未成形。非常感谢所有人。

我想感谢 John Haydu 和 Oracle 公司的 `MODEL` 子句开发团队，他们花时间检查了我为 O'Reilly 写的关于 `MODEL` 子句的文章，最终他们帮助我更深刻地理解了 `MODEL` 子句的工作原理。我想感谢 Oracle 公司的 Tom Kyte，他允许我将他的 `TO_BASE` 函数改写成一个纯 SQL 解决方案。微软公司的 Bruno Denuit 回答了我关于 SQL Server 2005 引入的窗口函数的问题。PostgreSQL 的 Simon Riggs 告诉了我 PostgreSQL 中的 SQL 新特性（非常感谢你，Simon。你让我知道哪些新特性会在什么时间发布出来，让我能在本书中收录其中的一部分，比如非常酷的 `GENERATE_SERIES` 函数，我认为这个函数相较于数据透视表提供了更优雅的解决方案）。

最后，同样重要的是，我想感谢 Kay Young。当你才华横溢、对自己的工作充满激情的时候，你发现了同样才华横溢、充满激情的人，和这样的人一起工作简直太棒了。你在本书中看到的许多实例都来自于 Wireless Generation 公司每天遇到的实际问题。Kay 是我的同事，我们一起创造了许多解决方案。Kay，我想感谢你，并让你知道我十分感激你在整个过程中给我的帮助。你给我提建议，帮我做语法纠错，还帮我写代码，你在本书的写作过程中发挥了至关重要的作用。和你一起工作太棒了，因为你，Wireless Generation 会成为更好的公司。

——Anthony Molinaro
2005 年 9 月

第 1 章

检索记录

本章主要介绍基本的 SELECT 语句。充分理解这些基础知识十分重要，因为本章中的许多内容不仅会出现在后面更复杂的实例里，同时也是日常 SQL 操作的一部分。

1.1　检索所有行和列

1. 问题

你有一张表，并且想查看表中的所有数据。

2. 解决方案

用特殊符号“*”对该表执行 SELECT 查询。

```
1 select *
2   from emp
```

3. 讨论

在 SQL 中，符号“*”有着特殊含义。该符号使得查询语句返回指定表的所有列。由于没有指定 WHERE 子句，因此所有行都会被提取出来。你也可以使用另一种方法，列出表中的每一列。

```
select empno,ename,job,sal,mgr,hiredate,comm,deptno
  from emp
```

在交互式即席查询中，使用 SELECT * 会更加容易。然而，在编写程序代码时，最好具体指明每一列。虽然执行结果相同，但指明每一列让你能清楚地知道查询语句会返回哪些列。类似地，对于其他人而言，这样的查询语句也会更易于理解，因为他们可能不知道所要查询的表里包含哪些列。

1.2 筛选行

1. 问题

你有一张表，并且只想查看满足指定条件的行。

2. 解决方案

使用 WHERE 子句指明保留哪些行。例如，下面的语句将查找部门编号为 10 的所有员工。

```
1 select *
2   from emp
3  where deptno = 10
```

3. 讨论

可以使用 WHERE 子句来筛选出我们感兴趣的行。如果 WHERE 子句的表达式针对某一行的判定结果为真，那么就会返回该行的数据。

大多数数据库都支持常用的运算符，例如 =、<、>、<=、>=、! 和 <>。除此之外，你可能需要指定多个条件来筛选数据，这时就需要使用 AND、OR 和圆括号。下一个实例将讨论这一点。

1.3 查找满足多个查询条件的行

1. 问题

你想返回满足多个查询条件的行。

2. 解决方案

使用带有 OR 和 AND 条件的 WHERE 子句。例如，如果你想找出部门编号为 10 的所有员工、有业务提成的所有员工以及部门编号是 20 且工资低于 2000 美元的所有员工。

```
1 select *
2   from emp
3  where deptno = 10
4     or comm is not null
5     or sal <= 2000 and deptno=20
```

3. 讨论

你可以组合使用 AND、OR 和圆括号来筛选满足多个查询条件的行。在这个实例中，WHERE 子句找出了如下的数据。

- DEPTNO 等于 10，或
- COMM 不是 Null，或
- DEPTNO 等于 20 且工资不高于 2000 美元的员工。

圆括号里的查询条件被一起评估。例如，试想一下如果采用下面的做法，检索结果会发生什么样的变化。

```
select *
 from emp
```

```
where (     deptno = 10
        or comm is not null
        or sal <= 2000
      )
  and deptno=20
EMPNO ENAME  JOB      MGR HIREDATE      SAL       COMM DEPTNO
----- ------ ------ ----- ----------- ----- ---------- ------
 7369 SMITH  CLERK   7902 17-DEC-1980   800                20
 7876 ADAMS  CLERK   7788 12-JAN-1983  1100                20
```

1.4 筛选列

1. 问题

你有一张表，并且只想查看特定列的值。

2. 解决方案

指定你感兴趣的列。例如，只查看员工的名字、部门编号和工资。

```
1 select ename,deptno,sal
2   from emp
```

3. 讨论

在 SELECT 语句里指定具体的列名，可以确保查询语句不会返回无关的数据。当在整个网络范围内检索数据时，这样做尤为重要，因为它避免了把时间浪费在检索不需要的数据上。

1.5 创建有意义的列名

1. 问题

你可能想要修改检索结果的列名，使其更具可读性且更易于理解。考虑下面这个查询，它返回的是每个员工的工资和业务提成。

```
1 select sal,comm
2   from emp
```

sal 指的是什么？是 sale 的缩写吗？是人名吗？ comm 又是什么？是 communication 的缩写吗？显然，检索结果应该让人容易理解。

2. 解决方案

使用 AS 关键字，并以 original_name AS new_name 的形式来修改检索结果的列名。对于一些数据库而言，AS 不是必需的，但所有的数据库都支持这个关键字。

```
1 select sal as salary, comm as commission
2   from emp

SALARY  COMMISSION
------- ----------
    800
   1600        300
   1250        500
   2975
```

```
1250       1300
2850
2450
3000
5000
1500          0
1100
 950
3000
1300
```

3. 讨论

使用 AS 关键字重新命名查询所返回的列，即是**创建别名**。新的列名被称作**别名**。创建好的别名对于查询语句大有裨益，它能让查询结果更易于理解。

1.6 在WHERE子句中引用别名列

1. 问题

你已经为检索结果集创建了有意义的列名，并且想利用 WHERE 子句过滤掉部分行数据。但是，如果你尝试在 WHERE 子句中引用别名列，查询无法顺利执行。

```
select sal as salary, comm as commission
  from emp
 where salary < 5000
```

2. 解决方案

把查询包装为一个内嵌视图，这样就可以引用别名列了。

```
1 select *
2   from (
3 select sal as salary, comm as commission
4   from emp
5        ) x
6  where salary < 5000
```

3. 讨论

在这个简单的实例中，你可以不使用内嵌视图。在 WHERE 子句里直接引用 COMM 列和 SAL 列，也可以达到同样的效果。当你想在 WHERE 子句中引用下列内容时，这个解决方案告诉你该如何做。

- 聚合函数
- 标量子查询
- 窗口函数
- 别名

将含有别名列的查询放入内嵌视图，就可以在外层查询中引用别名列。为什么要这么做呢？ WHERE 子句会比 SELECT 子句先执行，就最初那个失败的查询例子而言，当 WHERE 子句被执行时，SALARY 和 COMMISSION 尚不存在。直到 WHERE 子句执行完毕，那些别名列才会生效。然而，FROM 子句会先于 WHERE 子句执行。如果把最初的那个查询放入一个 FROM 子句，

其查询结果会在最外层的 WHERE 子句开始之前产生，这样一来，最外层的 WHERE 子句就能“看见”别名列了。当表里的某些列没有被恰当命名的时候，这个技巧尤其有用。



在本例中，内嵌视图的别名为 X。并非所有数据库都需要给内嵌视图取别名，但对于某些数据库而言，确实必须如此。不过，所有的数据库都支持这一点。

1.7 串联多列的值

1. 问题

你想将多列的值合并为一列。例如，你想查询 EMP 表，并获得如下结果集。

```
CLARK WORKS AS A MANAGER
KING WORKS AS A PRESIDENT
MILLER WORKS AS A CLERK
```

然而，你需要的数据来自 EMP 表的 ENAME 列和 JOB 列。

```
select ename, job
  from emp
 where deptno = 10

ENAME      JOB
---------- ---------
CLARK      MANAGER
KING       PRESIDENT
MILLER     CLERK
```

2. 解决方案

使用数据库中的内置函数来串联多列的值。

DB2、Oracle 和 PostgreSQL

这些数据库把双竖线作为串联运算符。

```
1 select ename||' WORKS AS A '||job as msg
2   from emp
3  where deptno=10
```

MySQL

该数据库使用 CONCAT 函数。

```
1 select concat(ename, ' WORKS AS A ',job) as msg
2   from emp
3  where deptno=10
```

SQL Server

该数据库使用“+”作为串联运算符。

```
1 select ename + ' WORKS AS A ' + job as msg
2   from emp
3  where deptno=10
```

3. 讨论

使用 CONCAT 函数可以串联多列的值。在 DB2、Oracle 和 PostgreSQL 中，“||”是 CONCAT 函数的快捷方式，在 SQL Server 中则为“+”。

1.8 在SELECT语句里使用条件逻辑

1. 问题

你想在 SELECT 语句中针对查询结果值执行 IF-ELSE 操作。例如，你想生成类似这样的结果：如果员工的工资少于 2000 美元，就返回 UNDERPAID；如果超过 4000 美元就返回 OVERPAID；若介于两者之间则返回 OK。查询结果如下所示。

```
ENAME             SAL STATUS
---------- ---------- ---------
SMITH             800 UNDERPAID
ALLEN            1600 UNDERPAID
WARD             1250 UNDERPAID
JONES            2975 OK
MARTIN           1250 UNDERPAID
BLAKE            2850 OK
CLARK            2450 OK
SCOTT            3000 OK
KING             5000 OVERPAID
TURNER           1500 UNDERPAID
ADAMS            1100 UNDERPAID
JAMES             950 UNDERPAID
FORD             3000 OK
MILLER           1300 UNDERPAID
```

2. 解决方案

在 SELECT 语句里直接使用 CASE 表达式来执行条件逻辑。

```
1 select ename,sal,
2        case when sal <= 2000 then 'UNDERPAID'
3             when sal >= 4000 then 'OVERPAID'
4             else 'OK'
5        end as status
6   from emp
```

3. 讨论

CASE 表达式能对查询结果执行条件逻辑判断。你可以为 CASE 表达式的执行结果取一个别名，使结果集更有可读性。就本例而言，STATUS 就是 CASE 表达式执行结果的别名。ELSE 子句是可选的，若没有它，对于不满足测试条件的行，CASE 表达式会返回 Null。

1.9 限定返回行数

1. 问题

你想限定查询结果的行数。你不关心排序，任意 *n* 行都可以。

2. 解决方案

使用数据库的内置功能来控制返回的行数。

DB2

使用 FETCH FIRST 子句。

```
1 select *
2   from emp fetch first 5 rows only
```

MySQL 和 PostgreSQL

使用 LIMIT 子句。

```
1 select *
2   from emp limit 5
```

Oracle

对于 Oracle 而言，通过在 WHERE 子句中限制 ROWNUM 的值来获得指定行数的结果集。

```
1 select *
2   from emp
3  where rownum <= 5
```

SQL Server

使用 TOP 关键字限定返回行数。

```
1 select top 5 *
2   from emp
```

3. 讨论

许多数据库提供了类似 FETCH FIRST 和 LIMIT 这样的子句来指定查询结果的行数。Oracle 与此不同，你必须使用 ROWNUM 的函数，该函数会为结果集里的每一行指定一个行号（从 1 开始，逐渐增大）。

当你使用 ROWNUM<=5 限定只返回最初的 5 行数据时，会发生如下的事情。

(1) Oracle 执行查询。
(2) Oracle 取得第一行数据，并把它的行号定为 1。
(3) 已经超过第 5 行了吗？如果没有，Oracle 会返回当前行，因为当前的行号满足小于或等于 5 这一条件。如果已经超过，那么 Oracle 就不返回当前行。
(4) Oracle 取得下一行数据，并且将行号加 1（得到 2，然后得到 3，再然后得到 4，以此类推）。
(5) 返回第 3 步。

如上述处理过程所示，Oracle 会在取得某一行数据之后再为其编号，这是关键之处。很多 Oracle 开发人员试图只获取一行数据，比如指定 ROWNUM=5，希望只返回第 5 行。但是，同时使用 ROWNUM 和等式条件是不对的。以下是使用 ROWNUM=5 后实际发生的事情。

(1) Oracle 执行查询。
(2) Oracle 取得第一行数据，并把它的行号定为 1。

(3) 已经到第 5 行了吗？如果没有，那么 Oracle 会舍弃这一行，因为它不符合条件。如果是，那么 Oracle 会返回当前行。但是，行号永远不可能到 5！

(4) Oracle 取得下一行数据，并把它的行号定为 1。这是因为查询结果的第 1 行的行号必须是 1。

(5) 返回第 3 步。

深入理解这一过程，你会明白为什么通过指定等式条件 ROWNUM=5 来获取第 5 行会失败。如果你不先获取第 1 行到第 4 行，第 5 行从何而来？

你可能会注意到，ROWNUM=1 确实能得到第 1 行，这似乎与上述解释相矛盾。ROWNUM=1 运行正常的原因在于，Oracle 必须至少尝试一次读取，才能确定表里是否有记录。仔细阅读以上处理过程，用 1 替换 5，你就会理解为什么指定 ROWNUM=1 作为条件（为了返回一行）会成功。

1.10 随机返回若干行记录

1. 问题

你希望从表中获取特定数量的随机记录。修改下面的语句，以便连续执行查询并使结果集含有 5 行不同的数据。

```
select ename, job
  from emp
```

2. 解决方案

使用数据库的内置函数来随机生成查询结果。在 ORDER BY 子句里使用该内置函数可以实现查询结果的随机排序。最后要结合 1.9 节中的技巧从随机排序结果里获取限定数目的行。

DB2

把内置函数 RAND 和 ORDER BY、FETCH 结合使用。

```
1 select ename,job
2   from emp
3  order by rand() fetch first 5 rows only
```

MySQL

把内置函数 RAND 和 LIMIT、ORDER BY 结合使用。

```
1 select ename,job
2   from emp
3  order by rand() limit 5
```

PostgreSQL

把内置函数 RANDOM 和 LIMIT、ORDER BY 结合使用。

```
1 select ename,job
2   from emp
3  order by random() limit 5
```

Oracle

在内置包 DBMS_RANDOM 里可以找到 VALUE 函数，把该内置函数和 ORDER BY、内置函数 ROWNUM 结合使用。

```
1 select *
2   from (
3  select ename, job
4    from emp
6   order by dbms_random.value()
7        )
8   where rownum <= 5
```

SQL Server

同时使用内置函数 NEWID 和 TOP、ORDER BY 来返回一个随机结果集。

```
1 select top 5 ename,job
2   from emp
3  order by newid()
```

3. 讨论

ORDER BY 子句能够接受一个函数的返回值，并利用该值改变当前结果集的顺序。在本例中，所有查询都是在 ORDER BY 子句执行结束后才限定返回值的行数。看过 Oracle 的解决方案后，非 Oracle 用户可能会受到启发，因为 Oracle 的解决方案展示了（在理论上）其他数据库内部是如何实现该查询的。

不要误认为 ORDER BY 子句中的函数是数值常量，这一点很重要。如果 ORDER BY 子句使用数值常量，那么就需要按照 SELECT 列表里的顺序来排序。如果 ORDER BY 子句使用了函数，那么就需要按照该函数的返回值来排序，而函数返回的值是根据结果集里的每一行计算而来的。

1.11　查找Null值

1. 问题

你想查找特定列的值为 Null 的所有行。

2. 解决方案

要判断一个值是否为 Null，必须使用 IS Null。

```
1 select *
2   from emp
3  where comm is null
```

3. 讨论

Null 值不会等于或者不等于任何值，甚至不能与其自身作比较。因此，不能使用 = 或 != 来测试某一列的值是否为 Null。判断一行是否含有 Null，必须使用 IS Null。你也可以使用 IS NOT Null 来找到给定列的值不是 Null 的所有行。

1.12 把Null值转换为实际值

1. 问题

有一些行包含 Null 值，但是你想在返回结果里将其替换为非 Null 值。

2. 解决方案

使用 COALESCE 函数将 Null 值替代为实际值。

```
1 select coalesce(comm,0)
2   from emp
```

3. 讨论

需要为 COALESCE 函数指定一个或多个参数。该函数会返回参数列表里的第一个非 Null 值。在本例中，若 COMM 不为 Null，会返回 COMM 值，否则返回 0。

处理 Null 值时，最好利用数据库的内置功能。在许多情况下，你会发现有不止一个函数能解决本实例中的问题。COALESCE 函数只是恰好适用于所有的数据库。除此之外，CASE 也适用于所有数据库。

```
select case
       when comm is not null then comm
       else 0
       end
  from emp
```

尽管 CASE 也能把 Null 值转换成实际值，但 COALESCE 函数更方便、更简洁。

1.13 查找匹配项

1. 问题

你想返回匹配某个特定字符串或模式的行。考虑下面的查询及其结果集。

```
select ename, job
  from emp
 where deptno in (10,20)

ENAME      JOB
---------- ---------
SMITH      CLERK
JONES      MANAGER
CLARK      MANAGER
SCOTT      ANALYST
KING       PRESIDENT
ADAMS      CLERK
FORD       ANALYST
MILLER      CLERK
```

你想从编号为 10 和 20 的两个部门中找到名字中含有字母 I 或职位以 ER 结尾的人。

```
ENAME      JOB
---------- ---------
SMITH      CLERK
JONES      MANAGER
CLARK      MANAGER
KING       PRESIDENT
MILLER     CLERK
```

2. 解决方案

结合使用 LIKE 运算符和 SQL 通配符 %。

```
1 select ename, job
2   from emp
3  where deptno in (10,20)
4    and (ename like '%I%' or job like '%ER')
```

3. 讨论

被用于 LIKE 模式匹配操作时，运算符 % 可以匹配任意长度的连续字符。大多数 SQL 实现也提供了下划线（_）运算符，用于匹配单个字符。通过在字母 I 前后都加上 %，任何（在任意位置）出现 I 的字符串都会被检索出来。如果没有使用 % 把检索模式围起来，那么 % 的位置会影响查询结果。例如，为了找到以 ER 结尾的职位，就需要在 ER 的前面加上 %；如果是要找以 ER 开头的职位，那就应该在 ER 的后面加上 %。

第 2 章

查询结果排序

本章主要介绍如何使查询结果个性化。如果知道如何控制和修改结果集，你就能提供更具可读性、更有意义的数据。

2.1 以指定顺序返回查询结果

1. 问题

你想显示部门编号为 10 的员工的名字、职位和工资，并根据工资从低到高排序。你希望返回如下结果集。

```
ENAME      JOB              SAL
---------- --------- ----------
MILLER     CLERK           1300
CLARK      MANAGER         2450
KING       PRESIDENT       5000
```

2. 解决方案

使用 ORDER BY 子句。

```
1 select ename,job,sal
2   from emp
3  where deptno = 10
4  order by sal asc
```

3. 讨论

ORDER BY 子句可以对结果集排序。本实例针对 SAL 按照升序排列。默认情况下，ORDER BY 会做升序排列，因此 ASC 子句是可选项。相应地，也可以通过指定 DESC 执行降序排列。

```
select ename,job,sal
  from emp
 where deptno = 10
 order by sal desc

ENAME      JOB              SAL
---------- --------- ----------
KING       PRESIDENT       5000
CLARK      MANAGER         2450
MILLER     CLERK           1300
```

你也可以不指定用于排序的列名，而指定一个数值来指代该列。数值从 1 开始，从左向右匹配 SELECT 列表里的列，如下所示。

```
select ename,job,sal
  from emp
 where deptno = 10
 order by 3 desc

ENAME      JOB              SAL
---------- --------- ----------
KING       PRESIDENT       5000
CLARK      MANAGER         2450
MILLER     CLERK           1300
```

上述 ORDER BY 子句里的数字 3 对应着 SELECT 列表的第 3 列，即 SAL。

2.2 多字段排序

1. 问题

针对 EMP 表的数据，你想先按照 DEPTNO 升序排列，然后再按照 SAL 降序排列。你希望返回如下所示的结果集。

```
     EMPNO     DEPTNO        SAL ENAME      JOB
---------- ---------- ---------- ---------- ---------
      7839         10       5000 KING       PRESIDENT
      7782         10       2450 CLARK      MANAGER
      7934         10       1300 MILLER     CLERK
      7788         20       3000 SCOTT      ANALYST
      7902         20       3000 FORD       ANALYST
      7566         20       2975 JONES      MANAGER
      7876         20       1100 ADAMS      CLERK
      7369         20        800 SMITH      CLERK
      7698         30       2850 BLAKE      MANAGER
      7499         30       1600 ALLEN      SALESMAN
      7844         30       1500 TURNER     SALESMAN
      7521         30       1250 WARD       SALESMAN
      7654         30       1250 MARTIN     SALESMAN
      7900         30        950 JAMES      CLERK
```

2. 解决方案

在 ORDER BY 子句中列出不同的排序列，以逗号分隔。

```
1 select empno,deptno,sal,ename,job
2   from emp
3  order by deptno, sal desc
```

3. 讨论

ORDER BY 的执行顺序是从左到右的。如果使用 SELECT 列表项对应的位置序号来指定排序项，那么这个数字序号不能大于 SELECT 列表里的项目个数。一般而言，你也可以根据一个没有被包含在 SELECT 列表里的列来排序，但必须明确地指定列名。不过，如果你的查询语句里有 GROUP BY 或 DISTINCT，那么就不能按照 SELECT 列表之外的列进行排序。

2.3 依据子串排序

1. 问题

你想按照一个字符串的特定部分排列查询结果。例如，你希望从 EMP 表检索员工的名字和职位，并且按照职位字段的最后两个字符对检索结果进行排序。结果集应该像下面这样。

```
ENAME      JOB
---------- ---------
KING       PRESIDENT
SMITH      CLERK
ADAMS      CLERK
JAMES      CLERK
MILLER     CLERK
JONES      MANAGER
CLARK      MANAGER
BLAKE      MANAGER
ALLEN      SALESMAN
MARTIN     SALESMAN
WARD       SALESMAN
TURNER     SALESMAN
SCOTT      ANALYST
FORD       ANALYST
```

2. 解决方案

DB2、MySQL、Oracle 和 PostgreSQL

在 ORDER BY 子句里使用 SUBSTR 函数。

```
select ename,job
  from emp
 order by substr(job,length(job)-2)
```

SQL Server

在 ORDER BY 子句里使用 SUBSTRING 函数。

```
select ename,job
  from emp
 order by substring(job,len(job)-2,2)
```

3. 讨论

利用数据库中的子串函数，你可以很方便地按照一个字符串的任意部分来排序。要想按照一个字符串的最后两个字符排序，需要先找到该字符串的结尾处（即字符串的长度），然

后减去 2。这样，起始位置就是该字符串的倒数第 2 个字符。然后，你就可以截取从指定起始位置开始直到字符串结束的所有字符。SQL Server 的 SUBSTRING 函数略有不同，它要求提供第 3 个参数来指定需要截取几个字符。对于本实例而言，第 3 个参数既可以是 2，也可以是任何大于 2 的数字。

2.4　对含有字母和数字的列排序

1. 问题

你有混合了字母和数字的数据，希望按照字母部分或者数字部分来排序。考虑如下所示的视图。

```
create view V
as
select ename||' '||deptno as data
  from emp

select * from V

DATA
-------------
SMITH 20
ALLEN 30
WARD 30
JONES 20
MARTIN 30
BLAKE 30
CLARK 10
SCOTT 20
KING 10
TURNER 30
ADAMS 20
JAMES 30
FORD 20
MILLER 10
```

你希望以 DEPTNO 或 ENAME 作为排序项。若按照 DEPTNO 排序，会产生如下所示的结果集。

```
DATA
-------------
CLARK 10
KING 10
MILLER 10
SMITH 20
ADAMS 20
FORD 20
SCOTT 20
JONES 20
ALLEN 30
BLAKE 30
MARTIN 30
JAMES 30
TURNER 30
WARD 30
```

若按照 ENAME 排序，会产生如下所示的结果集。

```
DATA
---------
ADAMS 20
ALLEN 30
BLAKE 30
CLARK 10
FORD 20
JAMES 30
JONES 20
KING 10
MARTIN 30
MILLER 10
SCOTT 20
SMITH 20
TURNER 30
WARD 30
```

2. 解决方案

Oracle 和 PostgreSQL

使用函数 REPLACE 和 TRANSLATE 修改用于排序的字符串。

```
/* 按照DEPTNO排序 */

1 select data
2   from V
3  order by replace(data,
4           replace(
5         translate(data,'0123456789','##########'),'#',''),'')

/* 按照ENAME排序 */

1 select data
2   from emp
3  order by replace(
4           translate(data,'0123456789','##########'),'#','')
```

DB2

DB2 的隐式类型转换比 Oracle 和 PostgreSQL 更严格，因此在创建视图 V 的时候，要先将 DEPTNO 的类型转换为 CHAR。这种方法没有创建一个新视图，而是直接使用内嵌视图。DB2 中的 REPLACE 函数和 TRANSLATE 函数的使用方式与 Oracle 和 PostgreSQL 中的相同，只是 TRANSLATE 函数的参数顺序稍有不同。

```
/* 按照DEPTNO排序 */

1  select *
2    from (
3  selectename||' '||cast(deptno as char(2)) as data
4    from emp
5         )v
6   order by replace(data,
7            replace(
```

```
8          translate(data,'##########','0123456789'),'#',''),'')
```

```
/* 按照ENAME排序 */

1  select *
2    from (
3  selectename||' '||cast(deptno as char(2)) as data
4    from emp
5         )v
6   order by replace(
7            translate(data,'##########','0123456789'),'#','')
```

MySQL 和 SQL Server

这些数据库不支持 TRANSLATE 函数，因此不能提供针对本问题的解决方案。

3. 讨论

使用 TRANSLATE 函数和 REPLACE 函数删除每一行的数字或者字符后，就能方便地按照剩余的部分排序。上述示例代码里被传递给 ORDER BY 的值如下述的结果集所示。（以 Oracle 解决方案为例的原因是，这 3 种数据库使用了同样的技巧，唯一特别之处在于 DB2 的 TRANSLATE 函数的参数顺序略有不同。）

```
select data,
       replace(data,
       replace(
     translate(data,'0123456789','##########'),'#',''),'') nums,
       replace(
     translate(data,'0123456789','##########'),'#','') chars
  from V

DATA         NUMS   CHARS
------------ ------ ----------
SMITH 20     20     SMITH
ALLEN 30     30     ALLEN
WARD 30      30     WARD
JONES 20     20     JONES
MARTIN 30    30     MARTIN
BLAKE 30     30     BLAKE
CLARK 10     10     CLARK
SCOTT 20     20     SCOTT
KING 10      10     KING
TURNER 30    30     TURNER
ADAMS 20     20     ADAMS
JAMES 30     30     JAMES
FORD 20      20     FORD
MILLER 10    10     MILLER
```

2.5　排序时对Null值的处理

1. 问题

你想按照 EMP 表的 COMM 列对查询结果进行排序，但该字段可能为 Null。因此，你需要想个办法来指定是否应该将 Null 值排到后面。

```
ENAME             SAL       COMM
---------- ---------- ----------
TURNER           1500          0
ALLEN            1600        300
WARD             1250        500
MARTIN           1250       1400
SMITH             800
JONES            2975
JAMES             950
MILLER           1300
FORD             3000
ADAMS            1100
BLAKE            2850
CLARK            2450
SCOTT            3000
KING             5000
```

或者你希望把 Null 值放在前面。

```
ENAME             SAL       COMM
---------- ---------- ----------
SMITH             800
JONES            2975
CLARK            2450
BLAKE            2850
SCOTT            3000
KING             5000
JAMES             950
MILLER           1300
FORD             3000
ADAMS            1100
MARTIN           1250       1400
WARD             1250        500
ALLEN            1600        300
TURNER           1500          0
```

2. 解决方案

根据你希望的排序方式（以及你所使用的数据库管理系统如何处理 Null 值排序问题），你能够对可能为 Null 的列进行升序排列或者降序排列。

```
1 select ename,sal,comm
2   from emp
3  order by 3

1 select ename,sal,comm
2   from emp
3  order by 3 desc
```

这个解决方案表明，如果一个可能为 Null 的列含有非 Null 值，它们也会相应地被升序排列或降序排列；这与你的直觉可能相反。但是，如果你希望采用与非 Null 值列不同的方式来排列 Null 值，例如，你可能想把非 Null 值以升序排列或降序排列，而把全部 Null 值都放到最后面，那么你就要使用 CASE 表达式来动态调整排序项。

DB2、MySQL、PostgreSQL 和 SQL Server

使用 CASE 表达式标记 Null 值。该标记有两种可能的取值：一种代表 Null 值，另一种代表非 Null 值。一旦你做好了标记，只要简单地把它放进 ORDER BY 子句就行了。这样一来，你就能在不影响非 Null 值的情况下，方便地调整 Null 值的位置了。

```
/* 非Null值COMM升序排列,全部Null值放到最后面 */

1 select ename,sal,comm
2   from (
3 select ename,sal,comm,
4        case when comm is null then 0 else 1 end as is_null
5   from emp
6        )x
7  order by is_nulldesc,comm

ENAME    SAL       COMM
------ ----- ----------
TURNER  1500          0
ALLEN   1600        300
WARD    1250        500
MARTIN  1250       1400
SMITH    800
JONES   2975
JAMES    950
MILLER  1300
FORD    3000
ADAMS   1100
BLAKE   2850
CLARK   2450
SCOTT   3000
KING    5000

/* 非Null值COMM降序排列,全部Null值放到最后面 */

1  select ename,sal,comm
2    from (
3  select ename,sal,comm,
4         case when comm is null then 0 else 1 end as is_null
5    from emp
6         )x
7   order by is_nulldesc,commdesc

ENAME    SAL       COMM
------ ----- ----------
MARTIN  1250       1400
WARD    1250        500
ALLEN   1600        300
TURNER  1500          0
SMITH    800
JONES   2975
JAMES    950
MILLER  1300
FORD    3000
ADAMS   1100
```

```
BLAKE   2850
CLARK   2450
SCOTT   3000
KING    5000
```

```
/* 非Null值COMM升序排列,全部Null值放到最前面 */

1  selectename,sal,comm
2    from (
3  selectename,sal,comm,
4         case when comm is null then 0 else 1 end as is_null
5    from emp
6         )x
7   order by is_null,comm

ENAME     SAL       COMM
------ ----- ----------
SMITH    800
JONES   2975
CLARK   2450
BLAKE   2850
SCOTT   3000
KING    5000
JAMES    950
MILLER  1300
FORD    3000
ADAMS   1100
TURNER  1500          0
ALLEN   1600        300
WARD    1250        500
MARTIN  1250       1400
```

```
/* 非Null值COMM降序排列,全部Null值放到最前面 */

1  selectename,sal,comm
2    from (
3  selectename,sal,comm,
4         case when comm is null then 0 else 1 end as is_null
5    from emp
6         )x
7   order by is_null,comm desc

ENAME     SAL       COMM
------ ----- ----------
SMITH    800
JONES   2975
CLARK   2450
BLAKE   2850
SCOTT   3000
KING    5000
JAMES    950
MILLER  1300
FORD    3000
ADAMS   1100
MARTIN  1250       1400
```

```
WARD    1250        500
ALLEN   1600        300
TURNER  1500          0
```

Oracle

如果你使用的是 Oracle 8*i* 或者更早的版本，可以使用上述针对其他平台的解决方案。如果使用的是 Oracle 9*i* 及后续版本，则能使用针对 `ORDER BY` 子句的扩展语法 `NULLS FIRST` 和 `NULLS LAST` 来决定 `Null` 值应该排到前面还是后面，而无须考虑非 `Null` 值的排序方式。

```
/* 非Null值COMM升序排列,全部Null值放到最后面 */

1 select ename,sal,comm
2   from emp
3  order by comm nulls last

ENAME    SAL       COMM
------ ----- ----------
TURNER  1500          0
ALLEN   1600        300
WARD    1250        500
MARTIN  1250       1400
SMITH    800
JONES   2975
JAMES    950
MILLER  1300
FORD    3000
ADAMS   1100
BLAKE   2850
CLARK   2450
SCOTT   3000
KING    5000

/* 非Null值COMM降序排列,全部Null值放到最后面 */

1 select ename,sal,comm
2   from emp
3  order by commdesc nulls last

ENAME    SAL       COMM
------ ----- ----------
MARTIN  1250       1400
WARD    1250        500
ALLEN   1600        300
TURNER  1500          0
SMITH    800
JONES   2975
JAMES    950
MILLER  1300
FORD    3000
ADAMS   1100
BLAKE   2850
CLARK   2450
SCOTT   3000
KING    5000
```

```
/* 非Null值COMM升序排列,全部Null值放到最前面 */

1 select ename,sal,comm
2   from emp
3  order by comm nulls first

ENAME    SAL       COMM
------ ----- ----------
SMITH    800
JONES   2975
CLARK   2450
BLAKE   2850
SCOTT   3000
KING    5000
JAMES    950
MILLER  1300
FORD    3000
ADAMS   1100
TURNER  1500          0
ALLEN   1600        300
WARD    1250        500
MARTIN  1250       1400

/* 非Null值COMM降序排列,全部Null值放到最前面 */

1 select ename,sal,comm
2   from emp
3  order by commdesc nulls first

ENAME    SAL       COMM
------ ----- ----------
SMITH    800
JONES   2975
CLARK   2450
BLAKE   2850
SCOTT   3000
KING    5000
JAMES    950
MILLER  1300
FORD    3000
ADAMS   1100
MARTIN  1250       1400
WARD    1250        500
ALLEN   1600        300
TURNER  1500          0
```

3. 讨论

除非数据库管理系统提供了一种方式，它能够让你在无须修改非 Null 值数据的情况下方便地把 Null 值排到最前面或者最后面（像 Oracle 那样），否则你就得添加一个辅助列。



在写作本书之时，DB2 用户能够在窗口函数 OVER 子句的 ORDER BY 里使用 NULLS FIRST 和 NULLS LAST，不过该语法不适用于针对整个结果集的 ORDER BY 子句。

辅助列（只存在于查询语句里，而不存在于表中）的目的是，让你能够识别出 Null 值，并控制其排在最前面还是最后面。对于非 Oracle 解决方案的查询语句，其内嵌视图 X 会返回如下结果集。

```
select ename,sal,comm,
       case when comm is null then 0 else 1 end as is_null
  from emp

ENAME    SAL       COMM    IS_NULL
------ ----- ---------- ----------
SMITH    800                     0
ALLEN   1600        300          1
WARD    1250        500          1
JONES   2975                     0
MARTIN  1250       1400          1
BLAKE   2850                     0
CLARK   2450                     0
SCOTT   3000                     0
KING    5000                     0
TURNER  1500          0          1
ADAMS   1100                     0
JAMES    950                     0
FORD    3000                     0
MILLER  1300                     0
```

通过使用 IS_NULL 返回的值，你就能在不影响 COMM 排序的情况下，轻而易举地把全部 Null 值放到最前面或者最后面。

2.6 依据条件逻辑动态调整排序项

1. 问题

你希望按照某个条件逻辑来排序。例如，如果 JOB 等于 SALESMAN，就要按照 COMM 来排序；否则，按照 SAL 排序。你希望返回如下所示的结果集。

```
ENAME             SAL JOB             COMM
---------- ---------- --------- ----------
TURNER           1500 SALESMAN           0
ALLEN            1600 SALESMAN         300
WARD             1250 SALESMAN         500
SMITH             800 CLERK
JAMES             950 CLERK
ADAMS            1100 CLERK
MARTIN           1250 SALESMAN        1300
MILLER           1300 CLERK
CLARK            2450 MANAGER
BLAKE            2850 MANAGER
JONES            2975 MANAGER
SCOTT            3000 ANALYST
FORD             3000 ANALYST
KING             5000 PRESIDENT
```

2. 解决方案

在 ORDER BY 子句里使用 CASE 表达式。

```
1 select ename,sal,job,comm
2   from emp
3  order by case when job = 'SALESMAN' then comm else sal end
```

3. 讨论

可以利用 CASE 表达式来动态调整结果的排序方式。上述示例代码中传递给 ORDER BY 的值如下所示。

```
select ename,sal,job,comm,
       case when job = 'SALESMAN' then comm else sal end as ordered
  from emp
 order by 5

ENAME             SAL JOB             COMM    ORDERED
---------- ---------- --------- ---------- ----------
TURNER           1500 SALESMAN           0          0
ALLEN            1600 SALESMAN         300        300
WARD             1250 SALESMAN         500        500
SMITH             800 CLERK                       800
JAMES             950 CLERK                       950
ADAMS            1100 CLERK                      1100
MARTIN           1250 SALESMAN        1300       1300
MILLER           1300 CLERK                      1300
CLARK            2450 MANAGER                    2450
BLAKE            2850 MANAGER                    2850
JONES            2975 MANAGER                    2975
SCOTT            3000 ANALYST                    3000
FORD             3000 ANALYST                    3000
KING             5000 PRESIDENT                  5000
```

第 3 章

多表查询

本章介绍如何利用连接查询和集合运算来合并多个表中的数据。连接查询是 SQL 的基础，集合运算也非常重要。为了掌握后续各章介绍的更复杂的查询，你必须首先学习本章中的连接查询和集合运算。

3.1 叠加两个行集

1. 问题

你想返回保存在多个表中的数据，理论上需要将一个结果集叠加在另一个之上。这些表可以没有相同的键，但它们的列的数据类型必须相同。例如，你想显示 EMP 表里部门编号为 10 的员工的名字和部门编号，以及 DEPT 表中各个部门的名称和编号。你希望得到如下所示的结果集。

```
ENAME_AND_DNAME      DEPTNO
--------------- ----------
CLARK                    10
KING                     10
MILLER                   10
----------
ACCOUNTING               10
RESEARCH                 20
SALES                    30
OPERATIONS               40
```

2. 解决方案

使用集合运算 UNION ALL 合并多个表中的行。

```
1  select ename as ename_and_dname, deptno
2    from emp
```

```
3   where deptno = 10
4   union all
5  select '----------', null
6    from t1
7   union all
8  select dname, deptno
9    from dept
```

3. 讨论

UNION ALL 将多个表中的行并入一个结果集。对于所有的集合运算来说，SELECT 列表里的所有项目必须保持数目相同，且数据类型匹配。例如，下面的两个检索都将失败。

```
select deptno  |  select deptno, dname
  from dept    |    from dept
 union all     |   union
select ename   |  select deptno
  from emp     |    from emp
```

尤其需要注意的是，如果有重复项，UNION ALL 也将一并纳入。如果你希望过滤掉重复项，可以使用 UNION 运算符。例如，如果针对 EMP.DEPTNO 和 DEPT.DEPTNO 执行 UNION 操作，就只会返回如下所示的 4 行数据。

```
select deptno
  from emp
 union
select deptno
  from dept

    DEPTNO
----------
        10
        20
        30
        40
```

使用 UNION 而不是 UNION ALL，则很可能会进行一次排序操作，以便删除重复项。当处理大型结果集的时候要想到这一点。大体而言，使用 UNION 等同于针对 UNION ALL 的输出结果再执行一次 DISTINCT 操作，如下所示。

```
select distinct deptno
  from (
select deptno
  from emp
 union all
select deptno
  from dept )

    DEPTNO
----------
        10
        20
        30
        40
```

除非有必要，否则不要在查询中使用 DISTINCT 操作，同样的规则也适用于 UNION。除非有必要，否则不要用 UNION 代替 UNION ALL。

3.2 合并相关行

1. 问题

你想根据一个共同的列或者具有相同值的列做连接查询，并返回多个表中的行。例如，你想显示部门编号为 10 的全部员工的名字及其部门所在地，但这些数据分别存储在两个表里。你希望得到如下所示的结果集。

```
ENAME      LOC
---------- -------------
CLARK      NEW YORK
KING       NEW YORK
MILLER     NEW YORK
```

2. 解决方案

通过 DEPTNO 字段把 EMP 表和 DEPT 表连接起来。

```
1 select e.ename, d.loc
2   from emp e, dept d
3  where e.deptno = d.deptno
4    and e.deptno = 10
```

3. 讨论

这个解决方案是一个关于**连接查询**的例子。更准确地说，它是**内连接**中的**相等连接**。连接查询是一种把来自两个表的行合并起来的操作。对于相等连接而言，其连接条件依赖于某个相等条件（例如，一个表的部门编号和另一个表的部门编号相等）。内连接是最早的一种连接，它返回的每一行都包含了来自参与连接查询的各个表的数据。

理论上，连接操作首先会依据 FROM 子句里列出的表生成笛卡儿积（列出所有可能的行组合），如下所示。

```
select e.ename, d.loc,
       e.deptno as emp_deptno,
       d.deptno as dept_deptno
  from emp e, dept d
where e.deptno = 10

ENAME      LOC           EMP_DEPTNO DEPT_DEPTNO
---------- ------------- ---------- -----------
CLARK      NEW YORK              10          10
KING       NEW YORK              10          10
MILLER     NEW YORK              10          10
CLARK      DALLAS                10          20
KING       DALLAS                10          20
MILLER     DALLAS                10          20
CLARK      CHICAGO               10          30
KING       CHICAGO               10          30
MILLER     CHICAGO               10          30
```

```
CLARK      BOSTON                10          40
KING       BOSTON                10          40
MILLER     BOSTON                10          40
```

EMP 表里部门编号为 10 的全部员工与 DEPT 表的所有部门组合都被列出来了。然后，通过 WHERE 子句里的 e.deptno 和 d.deptno 做连接操作，限定了只有 EMP.DEPTNO 和 DEPT.DEPTNO 相等的行才会被返回。

```
select e.ename, d.loc,
       e.deptno as emp_deptno,
       d.deptno as dept_deptno
  from emp e, dept d
 where e.deptno = d.deptno
   and e.deptno = 10

ENAME      LOC           EMP_DEPTNO DEPT_DEPTNO
---------- ------------- ---------- -----------
CLARK      NEW YORK              10          10
KING       NEW YORK              10          10
MILLER     NEW YORK              10          10
```

另一种写法是利用显式的 JOIN 子句（INNER 关键字是可选项）。

```
select e.ename, d.loc
  from emp e inner join dept d
    on (e.deptno = d.deptno)
 where e.deptno = 10
```

如果你更喜欢在 FROM 子句里（而不是在 WHERE 子句里）写明连接逻辑，则可以使用 JOIN 子句。这两种风格都符合 ANSI 标准，本书涉及的关系数据库管理系统的最新版本也都支持它们。

3.3 查找两个表中相同的行

1. 问题

你想找出两个表中相同的行，但需要连接多列。例如，考虑如下所示的视图 V。

```
create view V
as
select ename,job,sal
  from emp
 where job = 'CLERK'

select * from V

ENAME      JOB              SAL
---------- --------- ----------
SMITH      CLERK            800
ADAMS      CLERK           1100
JAMES      CLERK            950
MILLER     CLERK           1300
```

视图 V 只包含职位是 CLERK 的员工，但并没有显示 EMP 表中所有可能的列。你想从 EMP 表

获取与视图 V 相匹配的全部员工的 EMPNO、ENAME、JOB、SAL 和 DEPTNO，并且希望得到如下所示的结果集。

```
   EMPNO ENAME      JOB              SAL     DEPTNO
-------- ---------- --------- ---------- ----------
    7369 SMITH      CLERK            800         20
    7876 ADAMS      CLERK           1100         20
    7900 JAMES      CLERK            950         30
    7934 MILLER     CLERK           1300         10
```

2. 解决方案

把多个表中所有必要的列都连接起来，以获得正确的结果。也可以使用集合运算 INTERSECT 来替代连接查询，并返回两个表的交集（相同的行）。

MySQL 和 SQL Server

使用多个条件把 EMP 表和视图 V 连接起来。

```
1 select e.empno,e.ename,e.job,e.sal,e.deptno
2   from emp e, V
3  where e.ename = v.ename
4    and e.job   = v.job
5    and e.sal   = v.sal
```

除此之外，也可以使用 JOIN 子句执行同样的连接查询。

```
1 select e.empno,e.ename,e.job,e.sal,e.deptno
2   from emp e join V
3     on (    e.ename = v.ename
4         and e.job   = v.job
5         and e.sal   = v.sal )
```

DB2、Oracle 和 PostgreSQL

针对 MySQL 和 SQL Server 的解决方案也适用于 DB2、Oracle 和 PostgreSQL。如果你希望从视图 V 查询数据，就需要使用该方案。

如果你不需要检索视图 V 的某些列，可以使用集合运算 INTERSECT 和谓词 IN。

```
1 select empno,ename,job,sal,deptno
2   from emp
3  where (ename,job,sal) in (
4   select ename,job,sal from emp
5   intersect
6   select ename,job,sal from V
7  )
```

3.讨论

当执行连接查询时，为了得到正确的结果，必须慎重考虑要把哪些列作为连接项。当参与连接的行集里的某些列可能有共同值，而其他列有不同值的时候，这一点尤为重要。

集合运算 INTERSECT 会返回两个行集的相同部分。在使用 INTERSECT 时，必须保证两个表里参与比较的项目数目是相同的，并且数据类型也是相同的。注意，当执行集合运算时，默认不会返回重复项。

3.4 查找只存在于一个表中的数据

1. 问题

你希望从一个表（可以称之为源表）里找出那些在某个目标表里不存在的值。例如，你想找出在 DEPT 表中存在而在 EMP 表里却不存在的部门编号（如果有的话）。在示例数据中，DEPT 表里 DEPTNO 为 40 的数据并不存在于 EMP 表里，因此结果集应该如下所示。

```
    DEPTNO
----------
        40
```

2. 解决方案

计算差集的函数对解决本问题尤其有用。DB2、PostgreSQL 和 Oracle 支持差集运算。如果你所使用的数据库管理系统没有提供差集函数，那么就要采用 MySQL 和 SQL Server 解决方案介绍的子查询技巧。

DB2 和 PostgreSQL

使用集合运算 EXCEPT。

```
1 select deptno from dept
2 except
3 select deptno from emp
```

Oracle

使用集合运算 MINUS。

```
1 select deptno from dept
2 minus
3 select deptno from emp
```

MySQL 和 SQL Server

使用子查询得到 EMP 表中所有的 DEPTNO，并将该结果传入外层查询，然后外层查询会检索 DEPT 表，找出没有出现在子查询结果里的 DEPTNO 值。

```
1 select deptno
2   from dept
3  where deptno not in (select deptno from emp)
```

3. 讨论

DB2 和 PostgreSQL

DB2 和 PostgreSQL 提供的内置函数使得该操作非常简单。EXCEPT 运算符获取第一个结果集的数据，然后从中删除第二个结果集的数据。这种运算非常像减法。

包括 EXCEPT 在内的集合运算符在使用上都有一些限制条件。参与运算的两个 SELECT 列表要有相同的数据类型和值个数。而且，EXCEPT 不返回重复项；并且 Null 值不会产生问题，这与 NOT IN 子查询不同（参考对 MySQL 和 SQL Server 的讨论）。EXCEPT 运算符会返回只存在于第一个查询（EXCEPT 前面的查询）结果里而不存在于第二个查询（EXCEPT 后面的查询）结果里的行。

Oracle

Oracle 解决方案除了集合运算符叫作 MINUS 而不是 EXCEPT，其他方面与 DB2 和 PostgreSQL 的解决方案相同。另外，上述解释也适用于 Oracle。

MySQL 和 SQL Server

这个子查询会获取 EMP 表中所有的 DEPTNO。外层查询会返回 DEPT 表中“不存在于”或“未被包含在”子查询结果集里的所有的 DEPTNO 值。

当你使用 MySQL 和 SQL Server 的解决方案时，需要考虑排除重复项。其他数据库基于 EXCEPT 或者 MINUS 的解决方案已经从结果集中排除了重复的行，确保每个 DEPTNO 只出现一次。当然，之所以能这样做，是因为示例数据中的 DEPTNO 是表的主键。如果 DEPTNO 不是主键，你可以使用 DISTINCT 来确保每个在 EMP 表里缺少的 DEPTNO 值只出现一次，如下所示。

```
select distinct deptno
  from dept
where deptno not in (select deptno from emp)
```

在使用 NOT IN 时，要注意 Null 值。考虑如下的表 NEW_DEPT。

```
create table new_dept(deptno integer)
insert into new_dept values (10)
insert into new_dept values (50)
insert into new_dept values (null)
```

如果你试着使用 NOT IN 子查询检索存在于 DEPT 表却不存在于 NEW_DEPT 表的 DEPTNO，会发现查不到任何值。

```
select *
  from dept
 where deptno not in (select deptno from new_dept)
```

DEPTNO 为 20、30 和 40 的数据虽然不在 NEW_DEPT 表中，却没被上述查询检索到。原因就在于 NEW_DEPT 表里有 Null 值。子查询会返回 3 行 DEPTNO，分别为 10、50 和 Null 值。IN 和 NOT IN 本质上是 OR 运算，由于 Null 值参与 OR 逻辑运算的方式不同，IN 和 NOT IN 将会产生不同的结果。考虑以下分别使用 IN 和 OR 的例子。

```
select deptno
  from dept
 where deptno in ( 10,50,null )

 DEPTNO
-------
     10

select deptno
  from dept
 where (deptno=10 or deptno=50 or deptno=null)

 DEPTNO
-------
     10
```

再来看看使用 NOT IN 和 NOT OR 的例子。

```
select deptno
  from dept
 where deptno not in ( 10,50,null )

( no rows )

select deptno
  from dept
 where not (deptno=10 or deptno=50 or deptno=null)

( no rows )
```

如你所见，条件 DEPTNO NOT IN (10, 50, NULL) 等价于：

```
not (deptno=10 or deptno=50 or deptno=null)
```

对于 DEPTNO 是 50 的情况，下面是这个表达式的展开过程。

```
not (deptno=10 or deptno=50 or deptno=null)
(false or false or null)
(false or null)
null
```

在 SQL 中，TRUE or NULL 的运算结果是 TRUE，但 FALSE or NULL 的运算结果却是 Null！一旦混入了 Null，结果就会一直保持为 Null（除非你使用实例 1.11 介绍的技巧特意测试是否含有 Null）。必须谨记，当使用 IN 谓词以及当执行 OR 逻辑运算的时候，你要想到是否会涉及 Null 值。

为了避免 NOT IN 和 Null 值带来的问题，需要结合使用 NOT EXISTS 和关联子查询。关联子查询指的是外层查询执行后获得的结果集会被内层子查询引用。下面的例子给出了一个免受 Null 值影响的替代方案（回到"问题"部分给出的那个原始查询语句）。

```
select d.deptno
  from dept d
 where not exists ( select null
                      from emp e
                     where d.deptno = e.deptno )

    DEPTNO
----------
        40
```

上述查询语句遍历并评估 DEPT 表的每一行。针对每一行，会有如下操作。

(1) 执行子查询并检查当前的部门编号是否存在于 EMP 表。要注意关联条件 D.DEPTNO = E.DEPTNO，它通过部门编号把两个表连接起来。

(2) 如果子查询有结果返回给外层查询，那么 EXISTS (...) 的评估结果是 TRUE，这样 NOT EXISTS (...) 就是 FALSE，如此一来，外层查询就会舍弃当前行。

(3) 如果子查询没有返回任何结果，那么 NOT EXISTS (...) 的评估结果是 TRUE，由此外层查询就会返回当前行（因为它是一个不存在于 EMP 表中的部门编号）。

把 EXISTS/NOT EXISTS 和关联子查询一起使用时，SELECT 列表里的项目并不重要，因此我在这个例子中用了 SELECT NULL，这是为了让你把注意力放到子查询的连接操作上，而非 SELECT 列表的项目上。

3.5 从一个表检索与另一个表不相关的行

1. 问题

两个表有相同的键，你想在一个表里查找与另一个表不相匹配的行。例如，你想找出哪些部门没有员工。结果集如下所示。

```
    DEPTNO DNAME          LOC
---------- -------------- -------------
        40 OPERATIONS     BOSTON
```

如果想要找到每一个员工就职的部门，需要基于 EMP 表和 DEPT 表的 DEPTNO 列进行相等连接查询。DEPTNO 是两个表都有的列。不幸的是，相等连接无法找到哪些部门没有员工。这是因为，针对 EMP 表和 DEPT 表做相等连接操作，将返回满足连接条件的所有行。相反，你只想从 DEPT 表里找出那些不满足连接条件的行。

本问题乍看起来和前一个实例相同，但其实它们之间有微妙的差别。不同之处在于，前一个实例仅仅返回了没有出现在 EMP 表中的部门编号。然而，本实例可以很方便地从 DEPT 表中获取其他列。

2. 解决方案

基于共同列把两个表连接起来，返回一个表的所有行，不论这些行在另一个表里是否存在匹配行。然后，只保留那些不匹配的行即可。

DB2、MySQL、PostgreSQL 和 SQL Server

使用外连接并过滤掉 Null 值（关键字 OUTER 是可选的）。

```
1 select d.*
2   from dept d left outer join emp e
3     on (d.deptno = e.deptno)
4  where e.deptno is null
```

Oracle

对于 Oracle 9*i* 及其后续版本，上述解决方案仍然适用。当然，你也可以使用 Oracle 专有的外连接语法。

```
1 select d.*
2   from dept d, emp e
3  where d.deptno = e.deptno (+)
4    and e.deptno is null
```

Oracle 8*i* 数据库及更早的版本只能使用上述专有语法（注意，圆括号里是 +）来完成外连接操作。

3. 讨论

这个解决方案使用了外连接,并且只保留不匹配的行。这种操作有时候被称为反连接(anti-join)。为了更好地理解反连接，我们先来看一下没有过滤掉 Null 值的结果集。

```
select e.ename, e.deptno as emp_deptno, d.*
  from dept d left join emp e
    on (d.deptno = e.deptno)

ENAME      EMP_DEPTNO     DEPTNO DNAME          LOC
---------- ---------- ---------- -------------- ------------
SMITH              20         20 RESEARCH       DALLAS
ALLEN              30         30 SALE           CHICAGO
WARD               30         30 SALES          CHICAGO
JONES              20         20 RESEARCH       DALLAS
MARTIN             30         30 SALES          CHICAGO
BLAKE              30         30 SALES          CHICAGO
CLARK              10         10 ACCOUNTING     NEW YORK
SCOTT              20         20 RESEARCH       DALLAS
KING               10         10 ACCOUNTING     NEW YORK
TURNER             30         30 SALES          CHICAGO
ADAMS              20         20 RESEARCH       DALLAS
JAMES              30         30 SALES          CHICAGO
FORD               20         20 RESEARCH       DALLAS
MILLER             10         10 ACCOUNTING     NEW YORK
                              40 OPERATIONS     BOSTON
```

注意，最后一行的 EMP.ENAME 和 EMP_DEPTNO 都是 Null 值。这是因为没有员工在编号为 40 的部门工作。该解决方案使用 WHERE 子句，只保留了 EMP_DEPTNO 是 Null 值的行（这样只留下 DEPT 表中无法与 EMP 表相匹配的行)。

3.6　新增连接查询而不影响其他连接查询

1. 问题

你已经有了一个查询语句，它可以返回你想要的数据。你需要一些额外信息，但当你试图获取这些信息的时候，却丢失了原有的查询结果集中的数据。例如，你想查找所有员工的信息，包括他们所在部门的位置，以及他们收到奖金的日期。针对这个问题，EMP_BONUS 表包含了如下数据。

```
select * from emp_bonus

     EMPNO RECEIVED          TYPE
---------- ----------- ----------
      7369 14-MAR-2005          1
      7900 14-MAR-2005          2
      7788 14-MAR-2005          3
```

最初，你使用如下所示的查询语句。

```
select e.ename, d.loc
  from emp e, dept d
 where e.deptno=d.deptno
```

```
ENAME      LOC
---------- -------------
SMITH      DALLAS
ALLEN      CHICAGO
WARD       CHICAGO
JONES      DALLAS
MARTIN     CHICAGO
BLAKE      CHICAGO
CLARK      NEW YORK
SCOTT      DALLAS
KING       NEW YORK
TURNER     CHICAGO
ADAMS      DALLAS
JAMES      CHICAGO
FORD       DALLAS
MILLER     NEW YORK
```

对于有奖金的员工，你希望把他们收到奖金的日期也添加到结果集里，但连接了 EMP_BONUS 表后得到的行数却比预期的要少，因为并非所有的员工都有奖金。

```
select e.ename, d.loc,eb.received
  from emp e, dept d, emp_bonus eb
 where e.deptno=d.deptno
   and e.empno=eb.empno

ENAME      LOC           RECEIVED
---------- ------------- -----------
SCOTT      DALLAS        14-MAR-2005
SMITH      DALLAS        14-MAR-2005
JAMES      CHICAGO       14-MAR-2005
```

而你希望得到如下所示的结果集。

```
ENAME      LOC           RECEIVED
---------- ------------- -----------
ALLEN      CHICAGO
WARD       CHICAGO
MARTIN     CHICAGO
JAMES      CHICAGO       14-MAR-2005
TURNER     CHICAGO
BLAKE      CHICAGO
SMITH      DALLAS        14-MAR-2005
FORD       DALLAS
ADAMS      DALLAS
JONES      DALLAS
SCOTT      DALLAS        14-MAR-2005
CLARK      NEW YORK
KING       NEW YORK
MILLER     NEW YORK
```

2.解决方案

使用外连接既能够获得额外信息，又不会丢失原有的信息。首先连接 EMP 表和 DEPT 表，得到全部员工和他们所在部门的位置。然后外连接 EMP_BONUS 表，如果某个员工有奖金，则

检索其收到奖金的日期。下面是 DB2、MySQL、PostgreSQL 以及 SQL Server 的查询语法。

```
1 select e.ename, d.loc, eb.received
2   from emp e join dept d
3     on (e.deptno=d.deptno)
4   left join emp_bonus eb
5     on (e.empno=eb.empno)
6  order by 2
```

对于 Oracle 9*i* 数据库及其后续版本，上述解决方案仍然适用。除此之外，对于 Oracle 8*i* 数据库及更早的版本，可以使用 Oracle 专有的外连接语法。

```
1 select e.ename, d.loc, eb.received
2   from emp e, dept d, emp_bonus eb
3  where e.deptno=d.deptno
4    and e.empno=eb.empno (+)
5  order by 2
```

也可以使用标量子查询（即把子查询放置在 SELECT 列表里）来模仿外连接操作。

```
1 select e.ename, d.loc,
2        (select eb.received from emp_bonus eb
3          where eb.empno=e.empno) as received
4   from emp e, dept d
5  where e.deptno=d.deptno
6  order by 2
```

标量子查询解决方案适用于所有数据库。

3. 讨论

外连接查询会返回一个表中的所有行，以及另一个表中与之匹配的行。上一个实例中也出现了这种连接操作。外连接之所以能够解决本问题，是因为它不会过滤掉任何应该被返回的行。上述外连接查询返回的行数和没有外连接时一样多。而且，如果有收到奖金的日期，它也会返回那个日期。

使用标量子查询是解决本问题的一种巧妙做法，因为不需要修改主查询中正确的连接操作。在不破坏当前结果集的情况下，标量子查询是为现有查询语句添加额外数据的好办法。当使用标量子查询时，必须确保它们返回的是标量值（单值）。如果 SELECT 列表里的子查询返回多行，那么查询将会出错。

4. 参考资料

关于如何解决 SELECT 列表里的子查询不能返回多行数据的问题，参见 14.10 节。

3.7 确定两个表是否有相同的数据

1. 问题

你想知道两个表或两个视图里是否有相同的数据（行数和值）。考虑如下所示的视图。

```
create view V
as
select * from emp where deptno != 10
```

```
 union all
select * from emp where ename = 'WARD'

select * from V

EMPNO ENAME      JOB         MGR HIREDATE      SAL   COMM DEPTNO
----- ---------- --------- ----- ----------- ----- ----- ------
7369 SMITH      CLERK      7902 17-DEC-1980   800           20
7499 ALLEN      SALESMAN   7698 20-FEB-1981  1600   300     30
7521 WARD       SALESMAN   7698 22-FEB-1981  1250   500     30
7566 JONES      MANAGER    7839 02-APR-1981  2975           20
7654 MARTIN     SALESMAN   7698 28-SEP-1981  1250  1300     30
7698 BLAKE      MANAGER    7839 01-MAY-1981  2850           30
7788 SCOTT      ANALYST    7566 09-DEC-1982  3000           20
7844 TURNER     SALESMAN   7698 08-SEP-1981  1500     0     30
7876 ADAMS      CLERK      7788 12-JAN-1983  1100           20
7900 JAMES      CLERK      7698 03-DEC-1981   950           30
7902 FORD       ANALYST    7566 03-DEC-1981  3000           20
7521 WARD       SALESMAN   7698 22-FEB-1981  1250   500     30
```

你希望确定该视图是否和 EMP 表有完全相同的数据。与员工 WARD 相关的数据有两行，这表明相应的解决方案不仅要找出来不同的数据，还要找到重复的数据。根据 EMP 表的数据，二者的不同之处包括 3 行部门编号为 10 的数据以及两行员工 WARD 的数据。你希望返回如下所示的结果集。

```
EMPNO ENAME      JOB         MGR HIREDATE      SAL   COMM DEPTNO CNT
----- ---------- --------- ----- ----------- ----- ----- ------ ---
 7521 WARD       SALESMAN   7698 22-FEB-1981  1250   500     30   1
 7521 WARD       SALESMAN   7698 22-FEB-1981  1250   500     30   2
 7782 CLARK      MANAGER    7839 09-JUN-1981  2450           10   1
 7839 KING       PRESIDENT       17-NOV-1981  5000           10   1
 7934 MILLER     CLERK      7782 23-JAN-1982  1300           10   1
```

2. 解决方案

使用求差集的函数（MINUS 或 EXCEPT，这取决于你使用的数据库管理系统）可以很容易地比较表中的数据。如果你所使用的数据库管理系统没有提供类似功能，则可以使用关联子查询。

DB2 和 PostgreSQL

使用集合运算 EXCEPT 和 UNION ALL 找出视图 V 和 EMP 表的不同之处。

```
 1  (
 2   select empno,ename,job,mgr,hiredate,sal,comm,deptno,
 3           count(*) as cnt
 4     from V
 5    group by empno,ename,job,mgr,hiredate,sal,comm,deptno
 6   except
 7   select empno,ename,job,mgr,hiredate,sal,comm,deptno,
 8          count(*) as cnt
 9     from emp
10    group by empno,ename,job,mgr,hiredate,sal,comm,deptno
11  )
12    union all
```

```
13  (
14   select empno,ename,job,mgr,hiredate,sal,comm,deptno,
15          count(*) as cnt
16     from emp
17    group by empno,ename,job,mgr,hiredate,sal,comm,deptno
18   except
19   select empno,ename,job,mgr,hiredate,sal,comm,deptno,
20          count(*) as cnt
21     from V
22    group by empno,ename,job,mgr,hiredate,sal,comm,deptno
23  )
```

Oracle

使用集合运算 MINUS 和 UNION ALL 找出视图 V 和 EMP 表的不同之处。

```
 1  (
 2   select empno,ename,job,mgr,hiredate,sal,comm,deptno,
 3          count(*) as cnt
 4     from V
 5    group by empno,ename,job,mgr,hiredate,sal,comm,deptno
 6    minus
 7   select empno,ename,job,mgr,hiredate,sal,comm,deptno,
 8          count(*) as cnt
 9     from emp
10    group by empno,ename,job,mgr,hiredate,sal,comm,deptno
11  )
12    union all
13  (
14   select empno,ename,job,mgr,hiredate,sal,comm,deptno,
15          count(*) as cnt
16     from emp
17    group by empno,ename,job,mgr,hiredate,sal,comm,deptno
18    minus
19   select empno,ename,job,mgr,hiredate,sal,comm,deptno,
20          count(*) as cnt
21     from v
22    group by empno,ename,job,mgr,hiredate,sal,comm,deptno
23  )
```

MySQL 和 SQL Server

使用关联子查询和 UNION ALL 找出那些存在于视图 V 而不存在于 EMP 表的数据，以及存在于 EMP 表而不存在于视图 V 的数据，并将它们合并起来。

```
 1   select *
 2     from (
 3   select e.empno,e.ename,e.job,e.mgr,e.hiredate,
 4          e.sal,e.comm,e.deptno, count(*) as cnt
 5     from emp e
 6    group by empno,ename,job,mgr,hiredate,
 7             sal,comm,deptno
 8          )e
 9    where not exists (
10   select null
11     from (
```

```
12   select v.empno,v.ename,v.job,v.mgr,v.hiredate,
13          v.sal,v.comm,v.deptno, count(*) as cnt
14     from v
15    group by empno,ename,job,mgr,hiredate,
16             sal,comm,deptno
17          )v
18    where v.empno    = e.empno
19      and v.ename    = e.ename
20      and v.job      = e.job
21      and v.mgr      = e.mgr
22      and v.hiredate = e.hiredate
23      and v.sal      = e.sal
24      and v.deptno   = e.deptno
25      and v.cnt      = e.cnt
26      and coalesce(v.comm,0) = coalesce(e.comm,0)
27   )
28    union all
29   select *
30     from (
31   select v.empno,v.ename,v.job,v.mgr,v.hiredate,
32          v.sal,v.comm,v.deptno, count(*) as cnt
33     from v
34    group by empno,ename,job,mgr,hiredate,
35             sal,comm,deptno
36          )v
37    where not exists (
38   select null
39     from (
40   select e.empno,e.ename,e.job,e.mgr,e.hiredate,
41          e.sal,e.comm,e.deptno, count(*) as cnt
42     from emp e
43    group by empno,ename,job,mgr,hiredate,
44             sal,comm,deptno
45          )e
46    where v.empno    = e.empno
47      and v.ename    = e.ename
48      and v.job      = e.job
49      and v.mgr      = e.mgr
50      and v.hiredate = e.hiredate
51      and v.sal      = e.sal
52      and v.deptno   = e.deptno
53      and v.cnt      = e.cnt
54      and coalesce(v.comm,0) = coalesce(e.comm,0)
55   )
```

3. 讨论

尽管使用了不同的方法，但上述解决方案的原理并无差别。

(1) 首先，找出存在于 EMP 表而不存在于视图 V 的行；

(2) 然后与存在于视图 V 而不存在于 EMP 表的行合并（UNION ALL）。

如果两个表完全相同，则不会返回任何数据。如果两个表有不同之处，那么将返回那些不同的行。在比较两个表的时候，比较容易的做法是，在比较数据之前先单独比较行数。下面是一个行数比较的简单示例，适用于所有数据库管理系统。

```
select count(*)
  from emp
 union
select count(*)
  from dept

COUNT(*)
--------
       4
      14
```

因为 UNION 子句会过滤掉重复项，所以如果两个表的行数相同，则只会返回一行数据。本例中返回了两行数据，这说明两个表中没有完全相同的数据。

DB2、Oracle 和 PostgreSQL

MINUS 和 EXCEPT 的作用相同，所以这里只讨论 EXCEPT。UNION ALL 前后的两个查询语句非常相似。因此，为了说明这个解决方案的原理，我们将直接执行位于 UNION ALL 前面的那个查询。执行第 1 行至第 11 行后产生的结果集如下所示。

```
(
  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
         count(*) as cnt
    from V
   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
  except
  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
         count(*) as cnt
    from emp
   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
)

EMPNO ENAME      JOB         MGR HIREDATE      SAL  COMM DEPTNO CNT
----- ---------- --------- ----- ----------- ----- ----- ------ ---
 7521 WARD       SALESMAN   7698 22-FEB-1981  1250   500     30   2
```

上述结果集显示从视图 V 中查询到了一行数据，该行数据要么不存在于 EMP 表，要么它在视图 V 中出现的次数与 EMP 表中的不一致。对于本例而言，查询找到了员工 WARD 的重复行。如果你仍然不理解该结果集是如何产生的，可以分别执行位于 EXCEPT 前后的两个查询。你会发现，两个结果集的不同之处仅仅在于视图 V 中员工 WARD 相关行的 CNT 值。

位于 UNION ALL 后面的查询语句执行了和 UNION ALL 前面的查询相反的操作。该查询找出了那些存在于 EMP 表而不存在于视图 V 的行。

```
(
  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
         count(*) as cnt
    from emp
   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
   minus
  select empno,ename,job,mgr,hiredate,sal,comm,deptno,
         count(*) as cnt
    from v
   group by empno,ename,job,mgr,hiredate,sal,comm,deptno
```

```
)

EMPNO ENAME      JOB         MGR HIREDATE      SAL  COMM DEPTNO CNT
----- ---------- --------- ----- ----------- ----- ----- ------ ---
 7521 WARD       SALESMAN   7698 22-FEB-1981  1250   500     30   1
 7782 CLARK      MANAGER    7839 09-JUN-1981  2450           10   1
 7839 KING       PRESIDENT       17-NOV-1981  5000           10   1
 7934 MILLER     CLERK      7782 23-JAN-1982  1300           10   1
```

上述两个结果集通过 UNION ALL 合并后即可得到最终的结果集。

MySQL 和 SQL Server

位于 UNION ALL 前后的两个查询语句非常相似。为了理解基于子查询的解决方案，我们直接执行 UNION ALL 前面的查询。下面的查询是第 1 行至第 27 行的内容。

```
select *
   from (
 select e.empno,e.ename,e.job,e.mgr,e.hiredate,
        e.sal,e.comm,e.deptno, count(*) as cnt
   from emp e
  group by empno,ename,job,mgr,hiredate,
           sal,comm,deptno
        ) e
  where not exists (
select null
  from (
select v.empno,v.ename,v.job,v.mgr,v.hiredate,
       v.sal,v.comm,v.deptno, count(*) as cnt
  from v
 group by empno,ename,job,mgr,hiredate,
          sal,comm,deptno
       ) v
  where v.empno    = e.empno
    and v.ename    = e.ename
    and v.job      = e.job
    and v.mgr      = e.mgr
    and v.hiredate = e.hiredate
    and v.sal      = e.sal
    and v.deptno   = e.deptno
    and v.cnt      = e.cnt
    and coalesce(v.comm,0) = coalesce(e.comm,0)
)

EMPNO ENAME      JOB         MGR HIREDATE      SAL  COMM DEPTNO CNT
----- ---------- --------- ----- ----------- ----- ----- ------ ---
 7521 WARD       SALESMAN   7698 22-FEB-1981  1250   500     30   1
 7782 CLARK      MANAGER    7839 09-JUN-1981  2450           10   1
 7839 KING       PRESIDENT       17-NOV-1981  5000           10   1
 7934 MILLER     CLERK      7782 23-JAN-1982  1300           10   1
```

注意，这里比较的不是 EMP 表和视图 V，而是内嵌视图 E 和内嵌视图 V。计算出每一行数据出现的次数，并作为查询结果的一列返回。我们要比较每一行的数据及其出现的次数。如果你还是不理解比较操作是如何执行的，不妨单独执行两个子查询。下一步是找出存在于内嵌视图 E 而不存在于内嵌视图 V 的所有行（包括 CNT）。该操作使用了关联子查询和 NOT

EXISTS。连接查询将确定哪些行是相同的，NOT EXISTS 则筛选出内嵌视图 E 中与连接查询结果不匹配的行。UNION ALL 后面的查询语句做了相反的操作，它找出了所有存在于内嵌视图 V 而不存在于内嵌视图 E 的行。

```
select *
   from (
select v.empno,v.ename,v.job,v.mgr,v.hiredate,
       v.sal,v.comm,v.deptno, count(*) as cnt
  from v
 group by empno,ename,job,mgr,hiredate,
          sal,comm,deptno
       ) v
  where not exists (
select null
  from (
select e.empno,e.ename,e.job,e.mgr,e.hiredate,
       e.sal,e.comm,e.deptno, count(*) as cnt
  from emp e
 group by empno,ename,job,mgr,hiredate,
          sal,comm,deptno
      ) e
 where v.empno    = e.empno
   and v.ename    = e.ename
   and v.job      = e.job
   and v.mgr      = e.mgr
   and v.hiredate = e.hiredate
   and v.sal      = e.sal
   and v.deptno   = e.deptno
   and v.cnt      = e.cnt
   and coalesce(v.comm,0) = coalesce(e.comm,0)
)

EMPNO ENAME      JOB         MGR HIREDATE      SAL  COMM DEPTNO CNT
----- ---------- --------- ----- ----------- ----- ----- ------ ---
 7521 WARD       SALESMAN   7698 22-FEB-1981  1250   500     30   2
```

最后，使用 UNION ALL 合并两个结果集，即可得到最终的结果集。



Ales Spetic 和 Jonathan Gennick 在 *Transact-SQL Cookbook* 一书中给出了另一种解决方案。请参考这本书第 2 章“Comparing Two Sets for Equality”一节。

3.8 识别并消除笛卡儿积

1. 问题

你想找出部门编号为 10 的所有员工的名字及其部门所在的城市。下面的查询返回的数据是错误的。

```
select e.ename, d.loc
  from emp e, dept d
 where e.deptno = 10
```

```
ENAME      LOC
---------- -------------
CLARK      NEW YORK
CLARK      DALLAS
CLARK      CHICAGO
CLARK      BOSTON
KING       NEW YORK
KING       DALLAS
KING       CHICAGO
KING       BOSTON
MILLER     NEW YORK
MILLER     DALLAS
MILLER     CHICAGO
MILLER     BOSTON
```

正确的结果集如下所示。

```
ENAME      LOC
---------- -------------
CLARK      NEW YORK
KING       NEW YORK
MILLER     NEW YORK
```

2. 解决方案

在 FROM 子句里对两个表执行连接查询，以得到正确的结果集。

```
1 select e.ename, d.loc
2   from emp e, dept d
3  where e.deptno = 10
4    and d.deptno = e.deptno
```

3. 讨论

先看一下 DEPT 表的数据。

```
select * from dept

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

我们看到，编号为 10 的部门位于纽约，因此如果查询结果不是纽约，那就出错了。上述那个错误的查询语句返回的结果行数是 FROM 子句里两个表的行数的乘积。对于该查询而言，依据 EMP 表的部门编号等于 10 这一过滤条件，将产生 3 行结果。但是，由于没有对 DEPT 表做条件过滤，因此 DEPT 表中的全部 4 行数据都将被返回。3 乘以 4 等于 12，因此上述错误的查询语句会返回 12 行数据。为了消除笛卡儿积，我们通常会用到 $n-1$ 法则，其中 n 代表 FROM 子句里表的个数，$n-1$ 则代表消除笛卡儿积所必需的连接查询的最少次数。依据表里有什么样的键以及基于哪些列来实现表之间的连接操作，有时候必要的连接查询次数可能会超过 $n-1$ 次，但是当我们编写查询语句的时候，$n-1$ 法则仍然是一个很好的指导原则。



若使用得当，笛卡儿积会很有用。这一方法被广泛运用于多种查询中。笛卡儿积常用于变换或展开（以及合并）结果集，生成一系列的值，以及模拟 loop 循环。

3.9 组合使用连接查询与聚合函数

1. 问题

你想执行一个聚合操作，但查询语句涉及多个表。你希望确保表之间的连接查询不会干扰聚合操作。例如，你希望计算部门编号为 10 的员工的工资总额以及奖金总和。因为有部分员工多次获得奖金，所以在 EMP 表和 EMP_BONUS 表连接之后再执行聚合函数 SUM，就会得出错误的计算结果。在这个问题中，EMP_BONUS 表里有如下数据。

```
select * from emp_bonus

EMPNO RECEIVED          TYPE
----- ----------- ----------
 7934 17-MAR-2005          1
 7934 15-FEB-2005          2
 7839 15-FEB-2005          3
 7782 15-FEB-2005          1
```

现在，考虑下面的查询语句，它返回了部门编号为 10 的所有员工的工资和奖金。BONUS 表中的 TYPE 列决定了奖金的数额。若 TYPE 值等于 1，则奖金为工资的 10%；若 TYPE 值等于 2，则奖金为工资的 20%；若 TYPE 值等于 3，则奖金为工资的 30%。

```
select e.empno,
       e.ename,
       e.sal,
       e.deptno,
       e.sal*case when eb.type = 1 then .1
                  when eb.type = 2 then .2
                  else .3
              end as bonus
  from emp e, emp_bonus eb
 where e.empno  = eb.empno
   and e.deptno = 10

   EMPNO ENAME             SAL     DEPTNO      BONUS
-------- ---------- ---------- ---------- ----------
    7934 MILLER           1300         10        130
    7934 MILLER           1300         10        260
    7839 KING             5000         10       1500
    7782 CLARK            2450         10        245
```

到目前为止，一切都很顺利。然而，如果你试图连接 EMP_BONUS 表并计算奖金总和，就会出错。

```
select deptno,
       sum(sal) as total_sal,
       sum(bonus) as total_bonus
```

```
  from (
select e.empno,
       e.ename,
       e.sal,
       e.deptno,
       e.sal*case when eb.type = 1 then .1
                  when eb.type = 2 then .2
                  else .3
              end as bonus
  from emp e, emp_bonus eb
 where e.empno  = eb.empno
   and e.deptno = 10
       )x
 group by deptno

DEPTNO  TOTAL_SAL TOTAL_BONUS
------ ---------- -----------
    10      10050        2135
```

尽管奖金总额（TOTAL_BONUS）是正确的，但工资总额（TOTAL_SAL）却是错误的。部门编号为 10 的所有员工的工资总额应该是 8750，如下所示。

```
select sum(sal) from emp where deptno=10

  SUM(SAL)
----------
      8750
```

为什么工资总额不对呢？这是因为连接查询导致某些行的 SAL 列出现了两次。考虑下面连接 EMP 表和 EMP_BONUS 表的查询语句。

```
select e.ename,
       e.sal
  from emp e, emp_bonus eb
 where e.empno  = eb.empno
   and e.deptno = 10

    ENAME             SAL
    ---------- ----------
    CLARK            2450
    KING             5000
    MILLER           1300
    MILLER           1300
```

现在就能很容易地看出来为什么工资总额是错误的了，因为 MILLER 的工资被统计了两次。你真正想要的结果集应该如下所示。

```
DEPTNO  TOTAL_SAL TOTAL_BONUS
------ ---------- -----------
    10       8750        2135
```

2. 解决方案

在连接查询里进行聚合运算时，必须十分小心才行。如果连接查询产生了重复行，通常有两种办法来使用聚合函数，而且可以避免得出错误的计算结果。一种方法是，调用聚合函

数时直接使用关键字 DISTINCT，这样每个值都会先去掉重复项再参与计算；另一种方法是，在进行连接查询之前先执行聚合运算（以内嵌视图的方式），这样可以避免错误的结果，因为聚合运算发生在连接查询之前。下面的解决方案使用了 DISTINCT。之后，我们将讨论在连接查询之前使用内嵌视图执行聚合运算的做法。

MySQL 和 PostgreSQL

使用 DISTINCT 计算工资总额。

```
 1  select deptno,
 2         sum(distinct sal) as total_sal,
 3         sum(bonus) as total_bonus
 4    from (
 5  select e.empno,
 6         e.ename,
 7         e.sal,
 8         e.deptno,
 9         e.sal*case when eb.type = 1 then .1
10                    when eb.type = 2 then .2
11                    else .3
12               end  as bonus
13    from emp e, emp_bonus eb
14   where e.empno  = eb.empno
15     and e.deptno = 10
16         ) x
17   group by deptno
```

DB2、Oracle 和 SQL Server

上述解决方案也适用于这些数据库。另外，它们还支持窗口函数 SUM OVER。

```
 1  select distinct deptno,total_sal,total_bonus
 2    from (
 3  select e.empno,
 4         e.ename,
 5         sum(distinct e.sal) over
 6         (partition by e.deptno) as total_sal,
 7         e.deptno,
 8         sum(e.sal*case when eb.type = 1 then .1
 9                        when eb.type = 2 then .2
10                        else .3 end) over
11         (partition by deptno) as total_bonus
12    from emp e, emp_bonus eb
13   where e.empno  = eb.empno
14     and e.deptno = 10
15         ) x
```

3.讨论

MySQL 和 PostgreSQL

本实例“问题”部分的第二个查询语句把 EMP 表和 EMP_BONUS 表连接起来，并返回了员工 MILLER 的两行数据，这是导致 EMP 表的工资总额出错的原因（MILLER 的工资被加了两次）。对应的解决办法是只计算不同的 EMP.SAL 值。下面的查询语句是另一种解决方案。首先计算部门编号为 10 的全部员工的工资总额，然后连接 EMP 表和 EMP_BONUS 表。下面的查

询语句适用于所有的关系数据库管理系统。

```
select d.deptno,
       d.total_sal,
       sum(e.sal*case when eb.type = 1 then .1
                      when eb.type = 2 then .2
                      else .3 end) as total_bonus
  from emp e,
       emp_bonus eb,
       (
select deptno, sum(sal) as total_sal
  from emp
 where deptno = 10
 group by deptno
       ) d
 where e.deptno = d.deptno
   and e.empno  = eb.empno
 group by d.deptno,d.total_sal

   DEPTNO  TOTAL_SAL  TOTAL_BONUS
--------- ---------- ------------
       10       8750         2135
```

DB2、Oracle 和 SQL Server

上面的另一种解决方案利用了窗口函数 SUM OVER。下面的查询语句来自该解决方案的第 3 行至第 14 行，返回的结果集如下。

```
select e.empno,
       e.ename,
       sum(distinct e.sal) over
       (partition by e.deptno) as total_sal,
       e.deptno,
       sum(e.sal*case when eb.type = 1 then .1
                      when eb.type = 2 then .2
                      else .3 end) over
       (partition by deptno) as total_bonus
  from emp e, emp_bonus eb
 where e.empno  = eb.empno
   and e.deptno = 10

EMPNO ENAME       TOTAL_SAL DEPTNO TOTAL_BONUS

----- ---------- ---------- ------ ----------
 7934 MILLER           8750     10        2135
 7934 MILLER           8750     10        2135
 7782 CLARK            8750     10        2135
 7839 KING             8750     10        2135
```

窗口函数 SUM OVER 被调用了两次，第一次调用针对指定的分区或者分组计算工资总额。在本例中，分区指的是编号为 10 的部门，该部门员工的工资总额是 8750。第二次调用 SUM OVER 针对同一个分区计算奖金总额。最终的结果集则是在去除了 TOTAL_SAL、DEPTNO 以及 TOTAL_BONUS 组合的重复项之后产生的。

3.10 组合使用外连接查询与聚合函数

1. 问题

本节的问题和 3.9 节的大致相同，只是略微修改了 EMP_BONUS 表的数据，使得部门编号为 10 的员工中只有部分人获得了奖金。考虑如下所示的 EMP_BONUS 表和查询语句，该查询（表面上）计算出了部门编号为 10 的员工的工资总额和奖金总额。

```
select * from emp_bonus

     EMPNO RECEIVED          TYPE
---------- ----------- ----------
      7934 17-MAR-2005          1
      7934 15-FEB-2005          2

select deptno,
       sum(sal) as total_sal,
       sum(bonus) as total_bonus
  from (
select e.empno,
       e.ename,
       e.sal,
       e.deptno,
       e.sal*case when eb.type = 1 then .1
                  when eb.type = 2 then .2
                  else .3 end as bonus
  from emp e, emp_bonus eb
 where e.empno  = eb.empno
   and e.deptno = 10
       )
 group by deptno

DEPTNO  TOTAL_SAL TOTAL_BONUS
------ ---------- -----------
    10       2600         390
```

奖金总额的结果是正确的，但工资总额却不是部门编号为 10 的员工的工资总额。下面的查询语句解释了为什么工资总额不正确。

```
select e.empno,
       e.ename,
       e.sal,
       e.deptno,
       e.sal*case when eb.type = 1 then .1
                  when eb.type = 2 then .2
            else .3 end as bonus
  from emp e, emp_bonus eb
 where e.empno  = eb.empno
   and e.deptno = 10

   EMPNO ENAME             SAL     DEPTNO      BONUS
-------- ---------- ---------- ---------- ----------
    7934 MILLER           1300         10        130
    7934 MILLER           1300         10        260
```

上述查询没有计算部门编号为 10 的全部员工的工资总额，实际上只有 MILLER 的工资被计入总和，而且被错误地计算了两次。其实，你最终想得到如下所示的结果集。

```
DEPTNO  TOTAL_SAL TOTAL_BONUS
------ ---------- -----------
    10       8750         390
```

2. 解决方案

下面的解决方案也和 3.9 节的类似，不同之处在于要外连接 EMP_BONUS 表，确保把部门编号为 10 的全部员工都包括进来。

DB2、MySQL、PostgreSQL 和 SQL Server

外连接 EMP_BONUS 表，然后去掉部门编号为 10 的员工的重复项，再计算工资总和。

```
 1 select deptno,
 2        sum(distinct sal) as total_sal,
 3        sum(bonus) as total_bonus
 4   from (
 5 select e.empno,
 6        e.ename,
 7        e.sal,
 8        e.deptno,
 9        e.sal*case when eb.type is null then 0
10                   when eb.type = 1 then .1
11                   when eb.type = 2 then .2
12                   else .3 end as bonus
13   from emp e left outer join emp_bonus eb
14     on (e.empno = eb.empno)
15  where e.deptno = 10
16        )
17  group by deptno
```

也可以使用窗口函数 SUM OVER。

```
 1 select distinct deptno,total_sal,total_bonus
 2   from (
 3 select e.empno,
 4        e.ename,
 5        sum(distinct e.sal) over
 6        (partition by e.deptno) as total_sal,
 7        e.deptno,
 8        sum(e.sal*case when eb.type is null then 0
 9                       when eb.type = 1 then .1
10                       when eb.type = 2 then .2
11                       else .3
12                  end) over
13        (partition by deptno) as total_bonus
14   from emp e left outer join emp_bonus eb
15     on (e.empno = eb.empno)
16  where e.deptno = 10
17        ) x
```

Oracle

对于 Oracle 9*i* 数据库及其后续版本，上述解决方案仍然适用。除此之外，我们也可以使

用 Oracle 专有的外连接语法。对于 Oracle 8*i* 数据库及更早的版本，只能使用该语法实现外连接。

```
 1  select deptno,
 2         sum(distinct sal) as total_sal,
 3         sum(bonus) as total_bonus
 4    from (
 5  select e.empno,
 6         e.ename,
 7         e.sal,
 8         e.deptno,
 9         e.sal*case when eb.type is null then 0
10                    when eb.type = 1 then .1
11                    when eb.type = 2 then .2
12                    else .3 end as bonus
13    from emp e, emp_bonus eb
14   where e.empno  = eb.empno (+)
15     and e.deptno = 10
16         )
17   group by deptno
```

与 DB2 及其他数据库类似，Oracle 8*i* 数据库也支持 SUM OVER 语法，但必须把上面的查询语句里出现的外连接改为 Oracle 专有的语法。

3. 讨论

本实例“问题”部分中的第二个查询语句连接了 EMP 表和 EMP_BONUS 表，却只返回了员工 MILLER 的两行数据，这是导致 EMP 表的工资总额计算出错的原因（部门编号为 10 的其他员工没有奖金，他们的工资没有被计入总和）。解决办法则是把 EMP 表外连接到 EMP_BONUS 表，这样一来，那些没有奖金的员工也会被计算进来。如果一个员工没有奖金，那么 EMP_BONUS 表中的 TYPE 就是 Null 值。注意到这一点非常重要，因为 CASE 语句部分已经在 3.9 节的基础上稍微有了变动。如果 EMP_BONUS 表中的 TYPE 为 Null 值，则 CASE 表达式会返回 0，这样就不会对总和产生影响。

下面的查询语句是另一种解决方案。首先计算部门编号为 10 的员工的工资总额，然后再连接 EMP 表和 EMP_BONUS 表（这样就避免了使用外连接）。下面的查询语句适用于所有的关系数据库管理系统。

```
select d.deptno,
       d.total_sal,
       sum(e.sal*case when eb.type = 1 then .1
                      when eb.type = 2 then .2
                      else .3 end)  as total_bonus
   from emp e,
        emp_bonus eb,
        (
 select deptno, sum(sal) as total_sal
   from emp
  where deptno = 10
  group by deptno
        ) d
  where e.deptno = d.deptno
```

```
   and e.empno  = eb.empno
 group by d.deptno,d.total_sal

 DEPTNO  TOTAL_SAL  TOTAL_BONUS
--------- ---------- -----------
      10       8750         390
```

3.11　从多个表中返回缺失值

1. 问题

你想从多个表中返回缺失值。找到存在于 DEPT 表而不存在于 EMP 表的数据（即没有员工的部门）需要使用外连接。考虑下面的查询语句，该查询返回了 DEPT 表中所有的 DEPTNO 和 DNAME，以及每个部门里全部员工的名字（如果这个部门有员工的话）。

```
select d.deptno,d.dname,e.ename
  from dept d left outer join emp e
    on (d.deptno=e.deptno)

    DEPTNO DNAME          ENAME
---------- -------------- ----------
        20 RESEARCH       SMITH
        30 SALES          ALLEN
        30 SALES          WARD
        20 RESEARCH       JONES
        30 SALES          MARTIN
        30 SALES          BLAKE
        10 ACCOUNTING     CLARK
        20 RESEARCH       SCOTT
        10 ACCOUNTING     KING
        30 SALES          TURNER
        20 RESEARCH       ADAMS
        30 SALES          JAMES
        20 RESEARCH       FORD
        10 ACCOUNTING     MILLER
        40 OPERATIONS
```

最后一行是 OPERATIONS 部门，这个部门虽然没有员工，却也出现在了查询结果中，这是因为 DEPT 表外连接了 EMP 表。现在假设有一个员工不属于任何部门，你将如何返回以上结果集，并且包含那个不属于任何部门的员工呢？换句话说，你希望在同一个查询语句里既外连接到 EMP 表又外连接到 DEPT 表。在创建了新的员工数据之后，第一次尝试可能如下所示。

```
insert into emp (empno,ename,job,mgr,hiredate,sal,comm,deptno)
select 1111,'YODA','JEDI',null,hiredate,sal,comm,null
  from emp
 where ename = 'KING'

select d.deptno,d.dname,e.ename
  from dept d right outer join emp e
    on (d.deptno=e.deptno)
```

```
    DEPTNO DNAME          ENAME
---------- -------------- ----------
        10 ACCOUNTING     MILLER
        10 ACCOUNTING     KING
        10 ACCOUNTING     CLARK
        20 RESEARCH       FORD
        20 RESEARCH       ADAMS
        20 RESEARCH       SCOTT
        20 RESEARCH       JONES
        20 RESEARCH       SMITH
        30 SALES          JAMES
        30 SALES          TURNER
        30 SALES          BLAKE
        30 SALES          MARTIN
        30 SALES          WARD
        30 SALES          ALLEN
                          YODA
```

以上外连接查询包含了那个新的员工，却丢失了先前结果集里的 OPERATIONS 部门。最终的结果集应该既包括 YODA，也包括 OPERATIONS，如下所示。

```
    DEPTNO DNAME          ENAME
---------- -------------- ----------
        10 ACCOUNTING     CLARK
        10 ACCOUNTING     KING
        10 ACCOUNTING     MILLER
        20 RESEARCH       ADAMS
        20 RESEARCH       FORD
        20 RESEARCH       JONES
        20 RESEARCH       SCOTT
        20 RESEARCH       SMITH
        30 SALES          ALLEN
        30 SALES          BLAKE
        30 SALES          JAMES
        30 SALES          MARTIN
        30 SALES          TURNER
        30 SALES          WARD
        40 OPERATIONS
                           YODA
```

2. 解决方案

使用全外连接（full outer join），基于一个共同值从两个表中返回缺失值。

DB2、MySQL、PostgreSQL 和 SQL Server

使用显式的全外连接命令从两个表中返回缺失的行以及相匹配的行。

```
1 select d.deptno,d.dname,e.ename
2   from dept d full outer join emp e
3     on (d.deptno=e.deptno)
```

或者，也可以合并两个外连接的查询结果。

```
1 select d.deptno,d.dname,e.ename
2   from dept d right outer join emp e
```

```
3     on (d.deptno=e.deptno)
4  union
5 select d.deptno,d.dname,e.ename
6   from dept d left outer join emp e
7     on (d.deptno=e.deptno)
```

Oracle

对于 Oracle 9*i* 数据库及其后续版本，上述解决方案仍然适用。除此之外，我们也可以使用 Oracle 专有的外连接语法。对于 Oracle 8*i* 数据库及更早的版本，只能使用专有语法实现外连接。

```
1 select d.deptno,d.dname,e.ename
2   from dept d, emp e
3  where d.deptno = e.deptno(+)
4  union
5 select d.deptno,d.dname,e.ename
6   from dept d, emp e
7  where d.deptno(+) = e.deptno
```

3. 讨论

全外连接查询其实就是合并两个表的外连接查询的结果集。为了理解全外连接背后的运行原理，直接执行每一个外连接查询，然后合并其查询结果集即可。下面的查询找出了 DEPT 表里与 EMP 表相匹配的所有行（如果存在的话）。

```
select d.deptno,d.dname,e.ename
  from dept d left outer join emp e
    on (d.deptno = e.deptno)

DEPTNO DNAME          ENAME
------ -------------- ----------
    20 RESEARCH       SMITH
    30 SALES          ALLEN
    30 SALES          WARD
    20 RESEARCH       JONES
    30 SALES          MARTIN
    30 SALES          BLAKE
    10 ACCOUNTING     CLARK
    20 RESEARCH       SCOTT
    10 ACCOUNTING     KING
    30 SALES          TURNER
    20 RESEARCH       ADAMS
    30 SALES          JAMES
    20 RESEARCH       FORD
    10 ACCOUNTING     MILLER
    40 OPERATIONS
```

接下来的这个查询找出了 EMP 表里与 DEPT 表相匹配的所有行（如果存在的话）。

```
select d.deptno,d.dname,e.ename
  from dept d right outer join emp e
    on (d.deptno = e.deptno)

DEPTNO DNAME          ENAME
```

```
------ ------------- ----------
    10 ACCOUNTING    MILLER
    10 ACCOUNTING    KING
    10 ACCOUNTING    CLARK
    20 RESEARCH      FORD
    20 RESEARCH      ADAMS
    20 RESEARCH      SCOTT
    20 RESEARCH      JONES
    20 RESEARCH      SMITH
    30 SALES         JAMES
    30 SALES         TURNER
    30 SALES         BLAKE
    30 SALES         MARTIN
    30 SALES         WARD
    30 SALES         ALLEN
                     YODA
```

合并上面的两个查询结果，就可以得到最终的结果集。

3.12 在运算和比较中使用Null

1. 问题

Null 不会等于或不等于任何值，甚至不能与其自身进行比较，但是你希望对从 Null 列返回的数据进行评估，就像评估具体的值一样。例如，你想找出 EMP 表里业务提成（COMM 列）比员工 WARD 低的所有员工。检索结果应该包含业务提成为 Null 的员工。

2. 解决方案

使用如 COALESCE 这样的函数把 Null 转换为一个具体的、可以用于标准评估的值。

```
1 select ename,comm
2   from emp
3  where coalesce(comm,0) < ( select comm
4                               from emp
5                              where ename = 'WARD' )
```

3. 讨论

COALESCE 函数会返回参数列表里的第一个非 Null 值。就本实例而言，COMM 列中的 Null 会被替换为 0，这样才能与 WARD 的业务提成相比较。把 COALESCE 函数添加到 SELECT 列表，就能查看其执行结果。

```
select ename,comm,coalesce(comm,0)
  from emp
 where coalesce(comm,0) < ( select comm
                              from emp
                             where ename = 'WARD' )

ENAME            COMM COALESCE(COMM,0)
---------- ---------- ----------------
SMITH                                0
ALLEN             300              300
```

```
JONES                                    0
BLAKE                                    0
CLARK                                    0
SCOTT                                    0
KING                                     0
TURNER               0                   0
ADAMS                                    0
JAMES                                    0
FORD                                     0
MILLER                                   0
```

第 4 章

插入、更新和删除

前几章主要介绍基本的查询技巧，重点讨论如何从数据库获取数据。本章将讨论下面的三个话题。

- 插入新记录；
- 更新已有记录；
- 删除不需要的记录。

为了方便读者查阅，本章的实例已经按照话题进行分组。首先是与“插入”相关的实例，然后是与“更新”相关的实例，最后是与“删除”相关的实例。

插入记录通常简单易懂。它可以解决插入一行数据这样的简单问题。然而，在大多数情况下，使用基于集合的方法来创建新行，效率更高。为实现这一目标，你需要了解如何一次性地向数据库插入多行数据。

同样，更新和删除记录也很简单。你可以更新或者删除一条记录，也可以直接一次性地更新整个记录集。有许多种方便的方法可以删除记录。例如，你可以根据一个表中的某些行是否存在于另一个表中来删除这些行。

如果使用 SQL 数据库，你甚至可以一次性地插入、更新和删除所有相关记录，这是 SQL 新添加的标准功能。它现在看起来似乎不是很有用，不过 `MERGE` 语句的功能十分强大，能够将一个数据库表与另一个外部数据源同步。例如，一个来自远程系统的文本流。请参考本章的相关内容。

4.1 插入新记录

1. 问题

你希望向某个表中插入一条新记录。例如，你想插入一条新记录到 DEPT 表里。DEPTNO 的值应该为 50，DNAME 设为 PROGRAMMING，而 LOC 则是 BALTIMORE。

2. 解决方案

使用 INSERT 语句和 VALUES 子句可以一次插入一行。

```
insert into dept (deptno,dname,loc)
values (50,'PROGRAMMING','BALTIMORE')
```

对于 DB2 和 MySQL，你可以选择一次插入一行，或者通过附加多个 VALUES 列表来一次性插入多行记录。

```
/* 多行插入 */
insert into dept (deptno,dname,loc)
values (1,'A','B'),
       (2,'B','C')
```

3. 讨论

INSERT 语句允许你在数据库表里创建新的行。不论使用哪一种数据库，插入一条记录的语法都是相同的。

INSERT 还有一种简写方式，你可以省略字段列表。

```
insert into dept
values (50,'PROGRAMMING','BALTIMORE')
```

然而，如果你不指明目标列，则必须为所有列插入数据，还要注意 VALUES 列表的顺序。也就是说，你必须严格遵守 SELECT * 语句输出结果里各列的显示顺序。

4.2 插入默认值

1. 问题

可以定义表的某些列的默认值。你想在插入一行的时候使用预设的默认值，而不是指定的值。考虑下面的表。

```
create table D (id integer default 0)
```

你希望插入 0，并且不想显式地在 INSERT 语句的 VALUES 列表里指定 0。你只希望显式地插入默认值，而不管预设的默认值是什么。

2. 解决方案

所有数据库都支持使用 DEFAULT 关键字来显式地为某一列指定默认值，部分数据库还提供了其他方法来解决这一问题。

DEFAULT 关键字的使用示例如下。

```
insert into D values (default)
```

如果你没有为所有列都插入数值，则你需要直接指定列名。

```
insert into D (id) values (default)
```

Oracle 8*i* 数据库及更早的版本不支持 `DEFAULT` 关键字。因此，对于 Oracle 9*i* 之前的版本，没有办法为某一列显式地插入默认值。

如果所有列都预设了默认值，MySQL 允许指定一个空白的 `VALUES` 列表。

```
insert into D values ()
```

这样就可以为所有列创建预设的默认值。

PostgreSQL 和 SQL Server 支持 `DEFAULT VALUES` 子句。

```
insert into D default values
```

`DEFAULT VALUES` 子句会使得所有字段均取默认值。

3. 讨论

在创建表的过程中，`DEFAULT` 关键字会按照表定义中指定的默认值为特定的列插入一个值。`DEFAULT` 关键字适用于所有的数据库管理系统。

如果预先为表的每一列都定义了默认值（比如，本实例中的 `D` 表），则 MySQL、PostgreSQL 和 SQL Server 用户还有其他的选择。为了创建一行全是默认值的记录，你可以指定一个空白的 `VALUES` 列表（适用于 MySQL），或者使用 `DEFAULT VALUES` 子句（适用于 PostgreSQL 和 SQL Server）。否则，你就需要为表的每一列都指定 `DEFAULT` 关键字。

对于那些既有默认值列又有非默认值列的表，只要不把预设了默认值的列写入 `INSERT` 列表，就能方便地为它们插入默认值。也就是说，在这里你不需要使用 `DEFAULT` 关键字。假设 `D` 表还有一列，而该列没有预设的默认值。

```
create table D (id integer default 0, foo varchar(10))
```

在 `INSERT` 列表里只指定 `FOO` 列，就可以为 `ID` 列插入默认值。

```
insert into D (name) values ('Bar')
```

上述语句会产生一行 `ID` 列为 `0` 而 `FOO` 列为 `Bar` 的数据。`ID` 列会被设为默认值，因为我们没有为它指定其他的值。

4.3 使用Null覆盖默认值

1. 问题

你想要插入一列，该列有默认值，但你想将其设置为 `Null` 而不是默认值。考虑如下的表。

```
create table D (id integer default 0, foo VARCHAR(10))
```

你希望为 `ID` 列插入 `Null`。

2. 解决方案

在 `VALUES` 列表里显式地指定 `Null`。

```
insert into d (id, foo) values (null, 'Brighten')
```

3. 讨论

有些人不知道 INSERT 语句的 VALUES 列表可以显式指定 Null。一个典型的例子可以证明这一点，当不想为某一列插入特定值的时候，有些人会把该列从 VALUES 列表里去掉。

```
insert into d (foo) values ('Brighten')
```

在这里，并没有指定 ID 列的值。许多人可能以为 ID 列会被插入 Null，但是因为创建表的时候预设了默认值，所以上述 INSERT 语句执行后，ID 列会被设置为 0（默认值）。通过明确指定某字段为 Null，即使该字段有预设的默认值也能为其插入 Null。

4.4 复制数据到另一个表

1. 问题

你想使用查询语句把一些数据从一个表复制到另一个表里去。该查询语句可能很复杂，也可能很简单，但你希望最终把数据插入到另一个表。例如，你希望把 DEPT 表的部分数据复制到 DEPT_EAST 表。假设 DEPT_EAST 表已经被创建好了，其结构与 DEPT 表相同（有同样的列和数据类型），而且该表当前不含任何数据。

2. 解决方案

在 INSERT 语句后面附加一个用来检索目标数据的查询语句。

```
1 insert into dept_east (deptno,dname,loc)
2 select deptno,dname,loc
3   from dept
4  where loc in ( 'NEW YORK','BOSTON' )
```

3. 讨论

只需在 INSERT 后面附加一个用来检索目标数据的查询语句就可以解决这一问题。如果你希望复制表里的全部数据，那就要去掉 WHERE 子句。类似于正常的 INSERT 语句，你也不必明确指定要插入哪些列。但是，如果你选择不指明目标列，你就必须为所有列都插入数据，并且正如 4.1 节所讨论过的，你也必须注意 SELECT 列表里各列的顺序。

4.5 复制表定义

1. 问题

你想创建一个新表，该表和当前已存在的表保持相同的结构定义。例如，你希望为 DEPT 表创建一个副本，命名为 DEPT_2。但是，你只想复制它的表结构，而不复制数据。

2. 解决方案

DB2

使用 CREATE TABLE 语句和 LIKE 子句。

```
create table dept_2 like dept
```

Oracle、MySQL 和 PostgreSQL

使用 CREATE TABLE 语句和一个不返回任何数据的子查询。

```
1 create table dept_2
2 as
3 select *
4   from dept
5  where 1 = 0
```

SQL Server

使用 SELECT 语句和 INTO 子句，但要保证该查询不返回任何数据。

```
1 select *
2   into dept_2
3   from dept
4  where 1 = 0
```

3. 讨论

DB2

DB2 的 CREATE TABLE...LIKE 语句能以现有的表为模板快速创建一个新表。只要把模板表的名字放在 LIKE 关键字的后面即可。

Oracle、MySQL 和 PostgreSQL

使用 Create Table As Select（简写为 CTAS）语句时，除非为 WHERE 子句指定一个不可能为真的条件，否则，查询结果集将会被写入新表。本例中，WHERE 子句后面的表达式 1=0 会导致查询不返回任何结果。因此，上述 CTAS 语句的执行结果就是一张空表，该表的列取决于 SELECT 子句的查询结果。

SQL Server

使用 INTO 子句复制表定义时，除非为 WHERE 子句指定一个不可能为真的条件，否则的话查询结果集将会被写入新表。本例中，WHERE 子句后面的表达式 1=0 会导致查询不返回任何结果。上述语句的执行结果是一张空表，该表的列取决于 SELECT 子句的查询结果。

4.6 多表插入

1. 问题

你想把一个查询语句返回的结果集插入到多个目标表中。例如，你希望把 DEPT 表的数据分别插入到 DEPT_EAST 表、DEPT_WEST 表和 DEPT_MID 表。这 3 个表与 DEPT 表的结构相同（相同的列和数据类型），并且当前不含任何数据。

2. 解决方案

解决办法就是把查询结果插入到多个目标表中。与 4.4 节的不同之处在于，这次的目标表不止一个。

Oracle

使用 INSERT ALL 或者 INSERT FIRST 语句。除 ALL 和 FIRST 关键字不同之外，二者的语法并无二致。下面使用 INSERT ALL 语句，它能够确保兼顾所有目标表。

```
1 insert all
2   when loc in ('NEW YORK','BOSTON') then
3     into dept_east (deptno,dname,loc) values (deptno,dname,loc)
4   when loc = 'CHICAGO' then
5     into dept_mid  (deptno,dname,loc) values (deptno,dname,loc)
6   else
7     into dept_west (deptno,dname,loc) values (deptno,dname,loc)
8 select deptno,dname,loc
9   from dept
```

DB2

插入数据到一个内嵌视图，而该内嵌视图是对所有目标表执行 UNION ALL 得到的结果。你需要为各个目标表添加一个约束条件，确保每一行数据都会被插入到正确的表中。

```
create table dept_east
( deptno integer,
  dname  varchar(10),
  loc    varchar(10) check (loc in ('NEW YORK','BOSTON')))

create table dept_mid
( deptno integer,
  dname  varchar(10),
  loc    varchar(10) check (loc = 'CHICAGO'))

create table dept_west
( deptno integer,
  dname  varchar(10),
  loc    varchar(10) check (loc = 'DALLAS'))

1  insert into (
2    select * from dept_west union all
3    select * from dept_east union all
4    select * from dept_mid
5  ) select * from dept
```

MySQL、PostgreSQL 和 SQL Server

在写作本书时，这些数据库尚不支持多表插入。

3. 讨论

Oracle

Oracle 的多表插入使用 WHEN-THEN-ELSE 子句逐行评估嵌套 SELECT 语句所返回的结果，并将数据插入到相应的表中。虽然就本实例而言，INSERT ALL 和 INSERT FIRST 产生的结果相同，但它们仍然有差别。一旦 WHEN-THEN-ELSE 的结果为真，INSERT FIRST 会立即结束评估；INSERT ALL 则会逐一评估所有条件，而不论前面的测试结果是否真。因此，使用 INSERT ALL 有可能把同一行数据插入到多个表。

DB2

前面的 DB2 解决方案有点不太正统，因为它要求在目标表的定义里加入额外的约束条件，以确保查询结果的每一行都能被复制到正确的目标表中。该技巧的要点在于把数据插入一个针对所有目标表执行 UNION ALL 合并后得到的视图里去。如果为目标表添加的检查约束存在二义性（例如，多个表含有相同的检查约束），那么 INSERT 语句就无法判断要把数据

插入到哪个表，这样它就无法成功执行。

MySQL、PostgreSQL 和 SQL Server

在写作本书时，只有 Oracle 和 DB2 可以使用一个语句就能把某个查询的结果集插入到多个目标表。

4.7 禁止插入特定列

1. 问题

你想阻止用户或者错误的软件应用程序在某些列中插入数据。例如，你希望一个程序插入数据到 EMP 表，但只允许它插入 EMPNO、ENAME 和 JOB 列。

2. 解决方案

创建一个视图，只暴露那些你希望暴露的列。然后强制所有 INSERT 语句都被传送到该视图。

例如，下面的语句创建了一个视图，暴露出 EMP 表的三个列。

```
create view new_emps as
select empno, ename, job
  from emp
```

为那些可以向上面三个列中写入数据的用户和程序赋予视图的插入权限。不要把 EMP 表的插入权限授权给用户。这样用户插入数据到 NEW_EMPS 视图后就可以创建新的 EMP 记录，但是他们不能为视图定义里不存在的列提供插入值。

3. 讨论

向一个简单视图插入数据，数据库服务器会把它转换为针对基础表的插入操作。例如，下面的 INSERT 语句。

```
insert into new_emps
   (empno ename, job)
   values (1, 'Jonathan', 'Editor')
```

会被翻译成：

```
insert into emp
   (empno ename, job)
   values (1, 'Jonathan', 'Editor')
```

也可以插入数据到内嵌视图（目前只有 Oracle 支持），但这种做法可能不是很有用。

```
insert into
   (select empno, ename, job
      from emp)
values (1, 'Jonathan', 'Editor')
```

视图插入很复杂。如果不是针对最简单的视图做插入操作，那么问题会立刻变得超级复杂。如果想使用视图插入功能，那么就要仔细研读和全面理解相应的数据库关于此功能的帮助文档。

4.8　更新记录

1. 问题

你想更新一个表的部分记录或者全部记录。例如，你可能希望为部门编号为 20 的员工统一加薪 10%。下面的结果集显示了那个部门全部员工的 DEPTNO、ENAME 和 SAL。

```
select deptno,ename,sal
  from emp
 where deptno = 20
 order by 1,3

DEPTNO ENAME             SAL
------ ---------- ----------
    20 SMITH             800
    20 ADAMS            1100
    20 JONES            2975
    20 SCOTT            3000
    20 FORD             3000
```

你希望把所有 SAL 值都提高 10%。

2. 解决方案

使用 UPDATE 语句更新已有数据。例如：

```
1 update emp
2    set sal = sal*1.10
3  where deptno = 20
```

3. 讨论

使用 UPDATE 语句和 WHERE 子句来指定哪些行将被更新。如果省略 WHERE 子句，那么全部行都会被更新。上述解决方案里的表达式 SAL*1.10 返回增加 10% 后的工资。

在大规模数据更新之前，你可能希望先预览结果。可以通过提交一个 SELECT 语句，并把计划放入 SET 子句的表达式包含进 SELECT 语句来实现预览。下面的 SELECT 语句显示了工资增加 10% 后的结果。

```
select deptno,
       ename,
       sal      as orig_sal,
       sal*.10  as amt_to_add,
       sal*1.10 as new_sal
   from emp
  where deptno=20
  order by 1,5

DEPTNO ENAME  ORIG_SAL AMT_TO_ADD NEW_SAL
------ ------ -------- ---------- -------
    20 SMITH       800         80     880
    20 ADAMS      1100        110    1210
    20 JONES      2975        298    3273
    20 SCOTT      3000        300    3300
    20 FORD       3000        300    3300
```

工资的增加被分为两列：一列显示实际增加值，另一列显示增加后的工资。

4.9 当相关行存在时更新记录

1. 问题

你想更新一个表的部分行，而更新条件取决于另一个表中是否有与之相关的行。例如，如果一个员工出现在 EMP_BONUS 表中，你希望把他的工资（在 EMP 表中）上涨 20%。下面的结果集显示了 EMP_BONUS 表当前的数据。

```
select empno, ename
  from emp_bonus

     EMPNO ENAME
---------- ----------
      7369 SMITH
      7900 JAMES
      7934 MILLER
```

2. 解决方案

在 UPDATE 语句的 WHERE 子句里使用一个子查询来检索同时存在于 EMP 表和 EMP_BONUS 表的员工。这样 UPDATE 语句就能只检索那些员工的记录，并为其增加 20% 的工资。

```
1 update emp
2    set sal=sal*1.20
3  where empno in ( select empno from emp_bonus )
```

3.讨论

上述子查询的结果集代表了 EMP 表中将要被更新的行。IN 谓词用于评估 EMP 表中的 EMPNO 列是否存在于上述子查询返回的 EMPNO 列表里。如果是的话，相应的 SAL 值就会被更新。

除了 IN 谓词，也可以使用 EXISTS。

```
update emp
   set sal = sal*1.20
 where exists ( select null
                  from emp_bonus
                 where emp.empno=emp_bonus.empno )
```

你可能会惊讶于 EXISTS 子查询的 SELECT 列表只有一个 Null。不必担心，那个 Null 对更新操作没有负面影响。我认为这样做反而提高了查询语句的可读性，因为它强调了这样一个事实：真正决定更新操作的（例如，哪些行会被更新）是子查询里的 WHERE 子句，而不是 SELECT 列表。这与本解决方案里使用的 IN 谓词和子查询不同。

4.10 使用另一个表的数据更新记录

1. 问题

你想使用另一个表的值来更新当前的表。例如，现在有一个 NEW_SAL 表，存储了部分员工调整后的工资。NEW_SAL 表的数据如下。

```
select *
  from new_sal
```

```
DEPTNO        SAL
------ ----------
    10       4000
```

DEPTNO 列是 NEW_SAL 表的主键。你希望使用 NEW_SAL 表的数据来更新 EMP 表中部分员工的工资和业务提成。如果 EMP 表中的 DEPTNO 列和 NEW_SAL 表中的 DEPTNO 列相匹配，则将 EMP 表中的 SAL 列更新为 NEW_SAL 表中的 SAL 列，EMP 表中的 COMM 列更新为 NEW_SAL 表中 SAL 列的 50%。EMP 表的全部数据如下所示。

```
select deptno,ename,sal,comm
  from emp
 order by 1

DEPTNO ENAME             SAL       COMM
------ ---------- ---------- ----------
    10 CLARK            2450
    10 KING             5000
    10 MILLER           1300
    20 SMITH             800
    20 ADAMS            1100
    20 FORD             3000
    20 SCOTT            3000
    20 JONES            2975
    30 ALLEN            1600        300
    30 BLAKE            2850
    30 MARTIN           1250       1400
    30 JAMES             950
    30 TURNER           1500          0
    30 WARD             1250        500
```

2. 解决方案

把 NEW_SAL 表和 EMP 表连接起来，为 UPDATE 语句找出新的 COMM 值。正如本实例所示，在 UPDATE 语句里使用关联子查询是常用做法。另一种方法是创建一个视图（传统视图或者内嵌视图均可，这取决于数据库是否支持），然后更新该视图即可。

DB2 和 MySQL

使用关联查询来更新 EMP 表的 SAL 列和 COMM 列，同时也要使用另一个关联子查询来决定 EMP 表里有哪些行应该被更新。

```
1  update emp e set (e.sal,e.comm) = (select ns.sal, ns.sal/2
2                                      from new_sal ns
3                                     where ns.deptno=e.deptno)
4   where exists ( select null
5                    from new_sal ns
6                   where ns.deptno = e.deptno )
```

Oracle

DB2 的解决方案当然也适用于 Oracle，不过还有另外一种方法，即更新内嵌视图。

```
1 update (
2  select e.sal  as emp_sal, e.comm  as emp_comm,
```

```
3         ns.sal as ns_sal, ns.sal/2 as ns_comm
4    from emp e, new_sal ns
5   where e.deptno = ns.deptno
6  ) set emp_sal = ns_sal, emp_comm = ns_comm
```

PostgreSQL

DB2 的解决方案同样适用于 PostgreSQL，也可以在 UPDATE 语句里直接进行连接查询（非常方便）。

```
1 update emp
2    set sal  = ns.sal,
3        comm = ns.sal/2
4   from new_sal ns
5  where ns.deptno = emp.deptno
```

SQL Server

DB2 的解决方案同样适用于 SQL Server，也可以在 UPDATE 语句里直接进行连接查询（类似于 PostgreSQL 解决方案）。

```
1 update e
2    set e.sal  = ns.sal,
3        e.comm = ns.sal/2
4   from emp e,
5        new_sal ns
6  where ns.deptno = e.deptno
```

3. 讨论

在讨论各种解决方案之前，我想先说一下在 UPDATE 语句里使用 SELECT 查询提供新值的问题。在 UPDATE 语句的关联子查询里使用 WHERE 子句不同于针对需要更新的表所使用的 WHERE 子句。如果你看一下“解决方案”部分的 UPDATE 语句就会明白，EMP 表和 NEW_SAL 表基于 DEPTNO 列连接之后，把查询结果传递给了 UPDATE 语句的 SET 子句。对于部门编号为 10 的员工而言，会将对应的有效值传递给 SET 子句，因为在 NEW_SAL 表里有与之相匹配的 DEPTNO。但是对其他部门的员工而言，又当如何呢？ NEW_SAL 表里没有其他部门的数据，因此对于部门编号为 20 和 30 的员工来说，SAL 列和 COMM 列会变为 Null。除非使用 LIMIT、TOP 或者其他由数据库提供的限制结果集行数的机制，否则在 SQL 数据库里唯一能限制行数的办法就是使用 WHERE 子句。因此，为了正确地执行 UPDATE，有时候要针对需要更新的表使用 WHERE 子句，有时候却要在关联子查询里使用 WHERE 子句。

DB2 和 MySQL

为了确保不会误改 EMP 表的全部行，要记得在 UPDATE 语句的 WHERE 子句里使用关联子查询。因为仅仅在 SET 子句里执行连接查询（关联子查询）是不够的。UPDATE 语句的 WHERE 子句能够确保只更新 EMP 表中那些与 NEW_SAL 表的 DEPTNO 列相匹配的行。这适用于所有的关系数据库管理系统。

Oracle

Oracle 解决方案使用了可更新的连接视图，由相等连接查询来决定哪些行将被更新。我们可以单独执行该查询语句来确认哪些行会被更新，也可以单独执行该内嵌视图的查询语句来确认哪些行将被更新。若想正确地使用这种类型的 UPDATE，你必须先理解“键值保

持”（key-preservation）。DEPTNO 列是 NEW_SAL 表的主键，所以它的值是唯一的。当 EMP 表和 NEW_SAL 表做连接查询时，NEW_SAL 表的 DEPTNO 列在结果集里却不是唯一的，如下所示。

```
select e.empno, e.deptno e_dept, ns.sal, ns.deptno ns_deptno
   from emp e, new_sal ns
  where e.deptno = ns.deptno

EMPNO     E_DEPT        SAL  NS_DEPTNO
----- ---------- ---------- ----------
 7782         10       4000         10
 7839         10       4000         10
 7934         10       4000         10
```

为了让 Oracle 能够通过上述连接查询更新基础表，其中一个基础表必须符合键值保持的要求。也就是说，如果它的值在连接查询的结果集里不是唯一的，那么至少在基础表里是唯一的。在本例中，NEW_SAL 表的主键是 DEPTNO 列，因而符合唯一性的要求。既然 DEPTNO 列在基础表里是唯一的，那么它就可以多次出现在连接查询的结果集里，并被判定为符合键值保持的要求，更新操作也因此得以成功执行。

PostgreSQL 和 SQL Server

对于这两个数据库而言，语法略有不同，但方法是一样的。支持在 UPDATE 语句里直接进行连接查询自是方便之极。由于指明了要更新哪个表（通过在 UPDATE 关键字之后给出表的名字），数据库系统也就会知道要修改哪个表。除此之外，由于在更新操作里使用了连接查询（因为显式地使用了 WHERE 子句），我们得以避免一些关联子查询的陷阱。尤其是如果不小心漏掉了此处的连接查询，那么就能很容易发现问题。

4.11　合并记录

1. 问题

你想根据相关记录是否已经存在来插入、更新或删除一个表的记录。（如果记录存在，则更新它；如果不存在，则插入一条新记录；如果更新之后的记录不满足某个条件，则删除它。）例如，你希望按照如下的条件来修改 EMP_COMMISSION 表。

- 如果 EMP_COMMISSION 表的员工数据在 EMP 表里也存在相关记录，则更新其业务提成（COMM）为 1000。
- 对于所有可能会把 COMM 列更新为 1000 的员工，如果他们的 SAL 低于 2000，则删除相关记录（他们不应该存在于 EMP_ COMMISSION 表中）。
- 否则，就要从 EMP 表取出相应的 EMPNO、ENAME 和 DEPTNO，并插入 EMP_COMMISSION 表。

总之，你希望根据 EMP 表的给定行是否与 EMP_COMMISSION 表里的某条记录相匹配来决定要执行 UPDATE 语句还是 INSERT 语句。然后，如果 UPDATE 的结果导致某个员工的业务提成太高的话，你希望执行 DELETE 语句。

下面分别列出了当前 EMP 表和 EMP_COMMISSION 表里的数据。

```
select deptno,empno,ename,comm
  from emp
```

```
 order by 1

DEPTNO      EMPNO ENAME        COMM
------ ---------- ------ ----------
    10       7782 CLARK
    10       7839 KING
    10       7934 MILLER
    20       7369 SMITH
    20       7876 ADAMS
    20       7902 FORD
    20       7788 SCOTT
    20       7566 JONES
    30       7499 ALLEN         300
    30       7698 BLAKE
    30       7654 MARTIN       1400
    30       7900 JAMES
    30       7844 TURNER          0
    30       7521 WARD          500

select deptno,empno,ename,comm
  from emp_commission
 order by 1

    DEPTNO      EMPNO ENAME            COMM
---------- ---------- ---------- ----------
        10       7782 CLARK
        10       7839 KING
        10       7934 MILLER
```

2. 解决方案

Oracle 是目前仅有的只使用单个 SQL 语句就能解决本问题的关系数据库管理系统。这就是 MERGE 语句，在实际执行时，它会根据需要自动转换成相应的 UPDATE 语句或者 INSERT 语句，如下所示。

```
1  merge into emp_commission ec
2  using (select * from emp) emp
3     on (ec.empno=emp.empno)
4   when matched then
5        update set ec.comm = 1000
6        delete where (sal < 2000)
7   when not matched then
8        insert (ec.empno,ec.ename,ec.deptno,ec.comm)
9        values (emp.empno,emp.ename,emp.deptno,emp.comm)
```

3. 讨论

在上述的解决方案里，第 3 行的连接条件决定了哪些行已经存在，因而需要对它们执行更新操作。该连接查询是在 EMP_COMMISSION 表（别名为 EC）和子查询（别名为 emp）之间进行的。若连接操作成功，则相关的两行被认为是相匹配的，进而 WHEN MATCHED 子句里的 UPDATE 语句会被执行。同样，如果根据 EMPNO 列，EMP 表中的行在 EMP_COMMISSION 表里没有相匹配的行，则它会被插入到 EMP_COMMISSION 表中。在 EMP 表里，只有 DEPTNO 等于 10 的员工，其 EMP_COMMISSION 表的 COMM 才会被更新，而其他员工的数据都会被插入到 EMP_

COMMISSION 表。另外，员工 MILLER 的 DEPTNO 等于 10，本来他的业务提成（COMM）应该被更新，但是由于他的工资（SAL）低于 2000，因而他会被从 EMP_COMMISSION 表删除掉。

4.12　删除全表记录

1. 问题

你想删除某个表中的所有记录。

2. 解决方案

使用 DELETE 语句删除记录。例如，下述语句将把 EMP 表中的全部记录都删除掉。

```
delete from emp
```

3. 讨论

如果 DELETE 语句后面没有 WHERE 子句，则会删除指定表的全部记录。

4.13　删除指定记录

1. 问题

你想从一个表中删除满足特定条件的记录。

2. 解决方案

使用 DELETE 语句和 WHERE 子句，其中 WHERE 子句用于指定要删除的行。例如，下面的语句将删除部门编号为 10 的全部员工数据。

```
delete from emp where deptno = 10
```

3. 讨论

使用 DELETE 语句里的 WHERE 子句可以删除一个表的部分数据，而不是全部数据。

4.14　删除单行记录

1. 问题

你想删除表中的一行记录。

2. 解决方案

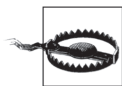
这是 4.13 节的特例。关键在于要保证你的检索条件能严苛到只返回需要被删除的那条记录。通常你需要按照主键删除记录。例如，如下语句将删除员工 CLARK（EMPNO 等于 7782）的数据。

```
delete from emp where empno = 7782
```

3. 讨论

删除数据的关键在于如何识别哪些行要被删除，DELETE 语句的影响力也取决于其 WHERE 子句。如果省略 WHERE 子句，则 DELETE 语句的影响范围会扩大至全表。通过在 WHERE 子句里

注明条件，我们能缩小范围到一组记录，甚至单行记录。如果要删除单行记录，通常你应该基于主键或者唯一键来识别要删除的记录。



如果删除条件基于主键或者唯一键，那么就能保证仅删除一条记录。（这是因为关系数据管理系统不允许两行数据含有相同的主键或者唯一键。）否则的话，就要先确保不会误删那些原本不该被删除的记录。

4.15 删除违反参照完整性的记录

1. 问题

你想从表里删除一些记录，因为在另一个表里不存在与这些记录相匹配的数据。例如，一些员工所属的部门其实并不存在，你希望删除这些员工。

2. 解决方案

使用 NOT EXISTS 谓词和子查询来确认部门编号的有效性。

```
delete from emp
 where not exists (
   select * from dept
    where dept.deptno = emp.deptno
)
```

或者，也可以使用 NOT IN 谓词。

```
delete from emp
where deptno not in (select deptno from dept)
```

3. 讨论

删除其实就是查询，最重要的步骤是要写出正确的 WHERE 子句条件，以找出要删除哪些记录。

上述 NOT EXISTS 解决方案使用关联子查询来检查给定的 EMP 记录是否存在一条与其 DEPTNO 列相匹配的 DEPT 记录。如果存在这样的记录，那么该 EMP 记录就应该被保留下来。否则，它就会被删除。每一条 EMP 记录都会被这样检查一次。

上述 IN 解决方案使用子查询来获取有效部门编号的列表。然后针对每一条 EMP 记录，都会与该列表做比照检查。如果一条 EMP 记录的 DEPTNO 不存在于该列表中，则该 EMP 记录会被删除。

4.16 删除重复记录

1. 问题

你想删除一个表里的重复记录，考虑如下的表。

```
create table dupes (id integer, name varchar(10))

insert into dupes values (1, 'NAPOLEON')
insert into dupes values (2, 'DYNAMITE')
```

```
insert into dupes values (3, 'DYNAMITE')
insert into dupes values (4, 'SHE SELLS')
insert into dupes values (5, 'SEA SHELLS')
insert into dupes values (6, 'SEA SHELLS')
insert into dupes values (7, 'SEA SHELLS')

select * from dupes order by 1

        ID NAME
---------- ----------
         1 NAPOLEON
         2 DYNAMITE
         3 DYNAMITE
         4 SHE SELLS
         5 SEA SHELLS
         6 SEA SHELLS
         7 SEA SHELLS
```

对于每一组重复的名字，例如 SEA SHELLS，你希望保留任意一个 ID，并删除其余的。不论删除 5 和 6，或者 5 和 7，或者 6 和 7，最终你只想要一条 SEA SHELLS 记录。

2. 解决方案

使用子查询和诸如 MIN 这样的聚合函数，任意选择并保留一个 ID（本例中 NAME 相同的情况下只有最小的 ID 会被保留）。

```
1  delete from dupes
2   where id not in ( select min(id)
3                       from dupes
4                      group by name )
```

3. 讨论

如果要删除重复记录，首先要明确两行数据在什么条件下才会被认为是“重复的记录”。就本实例而言，“重复的记录”是指它们的 NAME 列含有相同的值。确立了这样的定义之后，还要看一下能区分重复行的其他列，以便决定要保留的记录。最理想的状况是能区分重复行的那个列（或者那几列）是主键。在这里我选择了 ID 列，因为任意两条记录都不会有重复的 ID。

这个解决方案的关键之处在于按照重复值（本例中是 NAME）分组，并使用聚合函数找出一个将要保留的值。对于本解决方案而言，子查询会返回最小的 ID，其代表了不会被删除的行。

```
select min(id)
  from dupes
 group by name

    MIN(ID)
----------
         2
         1
         5
         4
```

然后，DELETE 语句会删除上述子查询返回值之外的所有 ID（本例中会删除的 ID 是 3、6 和 7）。为了能更好地理解上述子查询是如何运行的，不妨在 SELECT 列表里再加上 NAME 列。

```
select name, min(id)
  from dupes
 group by name

NAME          MIN(ID)
---------- ----------
DYNAMITE            2
NAPOLEON            1
SEA SHELLS          5
SHE SELLS           4
```

上述子查询返回的结果集就是将要被保留的行。DELETE 语句里的 NOT IN 谓词会删除其他行。

4.17　删除被其他表参照的记录

1. 问题

如果表里的一些记录会被其他表所参照的话，你想删除它们。考虑如下所示的 DEPT_ACCIDENTS 表，该表的每一行数据代表一起制造业生产事故。每一行都记录了发生事故的部门以及事故的类型。

```
create table dept_accidents
( deptno        integer,
  accident_name  varchar(20) )

insert into dept_accidents values (10,'BROKEN FOOT')
insert into dept_accidents values (10,'FLESH WOUND')
insert into dept_accidents values (20,'FIRE')
insert into dept_accidents values (20,'FIRE')
insert into dept_accidents values (20,'FLOOD')
insert into dept_accidents values (30,'BRUISED GLUTE')

select * from dept_accidents

    DEPTNO ACCIDENT_NAME
---------- --------------------
        10 BROKEN FOOT
        10 FLESH WOUND
        20 FIRE
        20 FIRE
        20 FLOOD
        30 BRUISED GLUTE
```

对于发生了 3 件以上事故的部门，你希望从 EMP 表里删除掉这些部门的全部员工记录。

2.解决方案

使用子查询和聚合函数 COUNT 找出发生过 3 次以上事故的部门，然后再删除在上述部门工作的员工。

```
1 delete from emp
2  where deptno in ( select deptno
3                       from dept_accidents
4                      group by deptno
5                     having count(*) >= 3 )
```

3. 讨论

下面的子查询用于识别哪些部门发生过 3 次以上事故。

```
select deptno
  from dept_accidents
 group by deptno
having count(*) >= 3

    DEPTNO
----------
        20
```

DELETE 语句会删除上述子查询返回的那些部门的全部员工（本例中只是部门编号等于 20 的部门）。

第 5 章

元数据查询

在本章的实例中，你可以查找关于**给定模式**（schema）的信息。例如，你可能想知道自己创建了哪些表，或者哪些外键没有添加索引。本书中的所有关系数据库管理系统都提供用于获取这些数据的表和视图。本章的实例将指导你从这些表和视图中获取信息。然而，这些实例没有讲解完所有的内容，请参考相关数据库的帮助文档，获取目录或数据字典视图（表）的完整信息。



为了方便演示，本章的所有实例都假设模式的名称为 SMEAGOL。

5.1　列举模式中的表

1. 问题

你想列出在某个模式里创建的所有表。

2. 解决方案

下面的每种解决方案都假设你正在使用 SMEAGOL 模式。每一种解决方案的基本思路都是一致的：检索数据库里的某个系统表（或者视图），你创建的每个表都对应着该系统表里的一行记录。

DB2

查询 `SYSCAT.TABLES`。

```
1 select tabname
2   from syscat.tables
3  where tabschema = 'SMEAGOL'
```

Oracle

查询 SYS.ALL_TABLES。

```
select table_name
  from all_tables
 where owner = 'SMEAGOL'
```

PostgreSQL、MySQL 和 SQL Server

查询 INFORMATION_SCHEMA.TABLES。

```
1 select table_name
2   from information_schema.tables
3  where table_schema = 'SMEAGOL'
```

3. 讨论

就像我们为自己的应用程序创建表和视图一样，数据库系统也以表和视图的形式把自身的信息提供给我们。例如，Oracle 数据库含有如 ALL_TABLES 这样的内容丰富的系统视图，可以查询关于表、索引、授权以及其他数据库对象的信息。



Oracle 数据库的目录视图其实就是普通的视图。它们基于一组底层表，但这些表中的信息非常不便于用户读取。目录视图为 Oracle 数据库的元数据提供了一个非常易用的外部接口。

Oracle 的系统视图和 DB2 的系统表各不相同。另外，PostgreSQL、MySQL 和 SQL Server 都支持信息模式（information schema），这是按照 ISO SQL 标准定义的一组视图。这就是为什么同一条查询语句能够适用于这三种数据库。

5.2　列举字段

1. 问题

你想列举一个表的列（即字段），以及它们的数据类型和在表中的位置。

2. 解决方案

下面的解决方案假设你希望列出 SMEAGOL 模式里的 EMP 表的各列，以及其数据类型和位置序号。

DB2

查询 SYSCAT.COLUMNS。

```
1 select colname, typename, colno
2   from syscat.columns
3  where tabname   = 'EMP'
4    and tabschema = 'SMEAGOL'
```

Oracle

查询 ALL_TAB_COLUMNS。

```
1  select column_name, data_type, column_id
2    from all_tab_columns
```

```
3   where owner      = 'SMEAGOL'
4     and table_name = 'EMP'
```

PostgreSQL、MySQL 和 SQL Server

查询 INFORMATION_SCHEMA.COLUMNS。

```
1 select column_name, data_type, ordinal_position
2   from information_schema.columns
3  where table_schema = 'SMEAGOL'
4    and table_name   = 'EMP'
```

3. 讨论

每一种数据库都提供了获取详细的列数据的方法。在本实例中，查询只返回了列名、数据类型和位置序号。除此之外，其他有用的信息还包括列长度、能否为 Null 以及默认值。

5.3　列举索引列

1. 问题

你想列出某个表的索引，包括构成索引的各列及其位置序号（如果有的话）。

2. 解决方案

下面的解决方案因数据库的不同而有所差异，但都假设你希望列出 SMEAGOL 模式中的 EMP 表的索引信息。

DB2

查询 SYSCAT.INDEXES。

```
1  select a.tabname, b.indname, b.colname, b.colseq
2    from syscat.indexes a,
3         syscat.indexcoluse b
4   where a.tabname   = 'EMP'
5     and a.tabschema = 'SMEAGOL'
6     and a.indschema = b.indschema
7     and a.indname   = b.indname
```

Oracle

查询 SYS.ALL_IND_COLUMNS。

```
select table_name, index_name, column_name, column_position
  from sys.all_ind_columns
 where table_name  = 'EMP'
   and table_owner = 'SMEAGOL'
```

PostgreSQL

查询 PG_CATALOG.PG_INDEXES 和 INFORMATION_SCHEMA.COLUMNS。

```
1  select a.tablename,a.indexname,b.column_name
2    from pg_catalog.pg_indexes a,
3         information_schema.columns b
4   where a.schemaname = 'SMEAGOL'
5     and a.tablename  = b.table_name
```

MySQL

使用 SHOW INDEX 命令。

```
show index from emp
```

SQL Server

查询 SYS.TABLES、SYS.INDEXES、SYS.INDEX_COLUMNS 和 SYS.COLUMNS。

```
 1  select a.name table_name,
 2         b.name index_name,
 3         d.name column_name,
 4         c.index_column_id
 5    from sys.tables a,
 6         sys.indexes b,
 7         sys.index_columns c,
 8         sys.columns d.
 9  where a.object_id = b.object_id
10    and b.object_id = c.object_id
11    and b.index_id  = c.index_id
12    and c.object_id = d.object_id
13    and c.column_id = d.column_id
14    and a.name      = 'EMP'
```

3. 讨论

说到查询，很重要的一点是要知道哪些列有索引。为那些经常被用来过滤数据并且相当有区分度的列添加索引，有利于提升查询的效果。索引对表之间的连接查询也非常有帮助。了解哪些列被加入了索引，能够有效地避免潜在的性能问题。此外，你可能希望查找关于索引自身的一些信息：遍历深度有多少级，有多少个不同的键，有多少个叶节点，等等。可以通过本实例中的解决方案所查询的视图和表来获取这类信息。

5.4 列举约束

1. 问题

你想列出模式中某个表的约束，以及与这些约束相关的列。例如，你希望找出 EMP 表的约束及相关的列。

2. 解决方案

DB2

查询 SYSCAT.TABCONST 和 SYSCAT.COLUMNS。

```
1  select a.tabname, a.constname, b.colname, a.type
2    from syscat.tabconst a,
3         syscat.columns b
4   where a.tabname   = 'EMP'
5     and a.tabschema = 'SMEAGOL'
6     and a.tabname   = b.tabname
7     and a.tabschema = b.tabschema
```

Oracle

查询 SYS.ALL_CONSTRAINTS 和 SYS.ALL_CONS_COLUMNS。

```
 1  select a.table_name,
           a.constraint_name,
 3         b.column_name,
 4         a.constraint_type
 5    from all_constraints a,
 6         all_cons_columns b
 7   where a.table_name      = 'EMP'
 8     and a.owner           = 'SMEAGOL'
 9     and a.table_name      = b.table_name
10     and a.owner           = b.owner
11     and a.constraint_name = b.constraint_name
```

PostgreSQL、MySQL 和 SQL Server

查询 INFORMATION_SCHEMA.TABLE_CONSTRAINTS 和 INFORMATION_SCHEMA.KEY_COLUMN_USAGE。

```
 1  select a.table_name,
 2         a.constraint_name,
 3         b.column_name,
 4         a.constraint_type
 5    from information_schema.table_constraints a,
 6         information_schema.key_column_usage b
 7   where a.table_name      = 'EMP'
 8     and a.table_schem     = 'SMEAGOL'
 9     and a.table_name      = b.table_name
10     and a.table_schema    = b.table_schema
11     and a.constraint_name = b.constraint_name
```

3. 讨论

毋庸置疑，约束是关系数据库的重要组成部分，我们甚至不需要解释为什么要列出一个表的约束。出于多个原因，列出一个表的约束是非常有用的。你可能希望找出那些没有主键的表，也可能想知道有哪些列应该被设置为外键却没有这么做（例如，子表的数据不同于父表，你希望知道这是怎么发生的），或者你可能希望了解检查约束。（一些列可以为 Null 吗？它们必须满足某些条件吗？）

5.5　列举非索引外键

1. 问题

你想列出含有非索引外键的表。例如，你希望确认 EMP 表的外键是否加入了索引。

2. 解决方案

DB2

查询 SYSCAT.TABCONST、SYSCAT.KEYCOLUSE、SYSCAT.INDEXES 和 SYSCAT.INDEXCOLUSE。

```
 1  select fkeys.tabname,
 2         fkeys.constname,
 3         fkeys.colname,
 4         ind_cols.indname
 5    from (
 6  select a.tabschema, a.tabname, a.constname, b.colname
```

```
 7   from syscat.tabconst a,
 8        syscat.keycoluse b
 9  where a.tabname   = 'EMP'
10    and a.tabschema = 'SMEAGOL'
11    and a.type      = 'F'
12    and a.tabname   = b.tabname
13    and a.tabschema = b.tabschema
14        ) fkeys
15        left join
16        (
17 select a.tabschema,
18        a.tabname,
19        a.indname,
20        b.colname
21   from syscat.indexes a,
22        syscat.indexcoluse b
23  where a.indschema = b.indschema
24    and a.indname   = b.indname
25        ) ind_cols
26     on (     fkeys.tabschema = ind_cols.tabschema
27          and fkeys.tabname   = ind_cols.tabname
28          and fkeys.colname   = ind_cols.colname )
29  where ind_cols.indname is null
```

Oracle

查询 SYS.ALL_CONS_COLUMNS、SYS.ALL_CONSTRAINTS 和 SYS.ALL_IND_COLUMNS。

```
 1 select a.table_name,
 2        a.constraint_name,
 3        a.column_name,
 4        c.index_name
 5   from all_cons_columns a,
 6        all_constraints b,
 7        all_ind_columns c
 8  where a.table_name        = 'EMP'
 9    and a.owner             = 'SMEAGOL'
10    and b.constraint_type   = 'R'
11    and a.owner             = b.owner
12    and a.table_name        = b.table_name
13    and a.constraint_name   = b.constraint_name
14    and a.owner             = c.table_owner  (+)
15    and a.table_name        = c.table_name   (+)
16    and a.column_name       = c.column_name  (+)
17    and c.index_name        is null
```

PostgreSQL

查询 INFORMATION_SCHEMA.KEY_COLUMN_USAGE、INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS、INFORMATION_SCHEMA.COLUMNS 和 PG_CATALOG.PG_INDEXES。

```
 1 select fkeys.table_name,
 2        fkeys.constraint_name,
 3        fkeys.column_name,
 4        ind_cols.indexname
 5   from (
```

```
 6  select a.constraint_schema,
 7         a.table_name,
 8         a.constraint_name,
 9         a.column_name
10    from information_schema.key_column_usage a,
11         information_schema.referential_constraints b
12   where a.constraint_name    = b.constraint_name
13     and a.constraint_schema  = b.constraint_schema
14     and a.constraint_schema  = 'SMEAGOL'
15     and a.table_name         = 'EMP'
16         ) fkeys
17         left join
18         (
19  select a.schemaname, a.tablename, a.indexname, b.column_name
20    from pg_catalog.pg_indexes a,
21         information_schema.columns b
22   where a.tablename  = b.table_name
23     and a.schemaname = b.table_schema
24         ) ind_cols
25      on (    fkeys.constraint_schema = ind_cols.schemaname
26          and fkeys.table_name        = ind_cols.tablename
27          and fkeys.column_name       = ind_cols.column_name )
28   where ind_cols.indexname is null
```

MySQL

使用 SHOW INDEX 命令获取诸如索引名称、索引列和列位置序号之类的索引信息。除此之外，我们还可以通过查询 INFORMATION_SCHEMA.KEY_COLUMN_USAGE 列出表的外键。对于 MySQL 5 而言，外键虽然默认是加入索引的，但事实上却可以被删掉。要确认外键列的索引是否已经被删除，可以针对特定的表执行 SHOW INDEX 命令，并比较其输出结果与 INFORMATION_SCHEMA.KEY_COLUMN_USAGE.COLUMN_NAME 的异同。如果 KEY_COLUMN_USAGE 里有对应的 COLUMN_NAME，但是 SHOW INDEX 输出的结果里却没有，那么就说明该列没有索引。

SQL Server

查询 SYS.TABLES、SYS.FOREIGN_KEYS、SYS.COLUMNS、SYS.INDEXES 和 SYS.INDEX_COLUMNS。

```
 1  select fkeys.table_name,
 2         fkeys.constraint_name,
 3         fkeys.column_name,
 4         ind_cols.index_name
 5    from (
 6  select a.object_id,
 7         d.column_id,
 8         a.name table_name,
 9         b.name constraint_name,
10         d.name column_name
11    from sys.tables a
12         join
13         sys.foreign_keys b
14      on (    a.name      = 'EMP'
15         and a.object_id = b.parent_object_id
16         )
17         join
18         sys.foreign_key_columns c
```

```
19      on ( b.object_id = c.constraint_object_id )
20         join
21         sys.columns d
22      on (   c.constraint_column_id = d.column_id
23         and a.object_id            = d.object_id
24         )
25         ) fkeys
26         left join
27         (
28  select a.name index_name,
29         b.object_id,
30         b.column_id
31    from sys.indexes a,
32         sys.index_columns b
33   where a.index_id = b.index_id
34          ) ind_cols
35      on (     fkeys.object_id = ind_cols.object_id
36           and fkeys.column_id = ind_cols.column_id )
37   where ind_cols.index_name is null
```

3. 讨论

当修改数据时，每一种数据库的锁机制都不尽相同。如果通过外键实现父子关系，那么为子表里对应的列加上索引有助于减少锁（详情请参考各数据库的帮助文档）。还有一种应用场景：子表和父表常用外键列做连接查询，因而加上索引有助于提升查询性能。

5.6 用SQL生成SQL

1. 问题

你想生成动态的 SQL 语句，例如你的目的是将某些维护任务自动化。你希望完成 3 项任务：计算各个表的行数，禁用各个表的外键约束，根据表里的数据生成插入脚本。

2. 解决方案

基本思路是使用字符串拼接 SQL 语句，通过查询某些表来获取需要填入的数据（例如数据库对象名称）。注意，这些查询仅生成 SQL 语句。你需要手动或者通过其他方式运行脚本，以执行这些 SQL 语句。下面的示例都是针对 Oracle 数据库的。对于其他数据库而言，做法应该极其相似，但是数据字典表的名称以及日期格式之类的细节或有不同。下面的输出结果来自我的笔记本电脑上的一个 Oracle 实例。你的计算机上的执行结果当然会有所不同。

```
/* 生成SQL以计算各个表的行数 */

select 'select count(*) from '||table_name||';' cnts
  from user_tables;

CNTS
--------------------------------------
select count(*) from ANT;
select count(*) from BONUS;
select count(*) from DEMO1;
```

```
select count(*) from DEMO2;
select count(*) from DEPT;
select count(*) from DUMMY;
select count(*) from EMP;
select count(*) from EMP_SALES;
select count(*) from EMP_SCORE;
select count(*) from PROFESSOR;
select count(*) from T;
select count(*) from T1;
select count(*) from T2;
select count(*) from T3;
select count(*) from TEACH;
select count(*) from TEST;
select count(*) from TRX_LOG;
select count(*) from X;

/* 禁用所有表的外键约束 */

select 'alter table '||table_name||
       ' disable constraint '||constraint_name||';' cons
   from user_constraints
  where constraint_type = 'R';

CONS
--------------------------------------------------------
alter table ANT disable constraint ANT_FK;
alter table BONUS disable constraint BONUS_FK;
alter table DEMO1 disable constraint DEMO1_FK;
alter table DEMO2 disable constraint DEMO2_FK;
alter table DEPT disable constraint DEPT_FK;
alter table DUMMY disable constraint DUMMY_FK;
alter table EMP disable constraint EMP_FK;
alter table EMP_SALES disable constraint EMP_SALES_FK;
alter table EMP_SCORE disable constraint EMP_SCORE_FK;
alter table PROFESSOR disable constraint PROFESSOR_FK;

/* 根据EMP表的某些列生成插入脚本 */

select 'insert into emp(empno,ename,hiredate) '||chr(10)||
       'values( '||empno||','||''''||ename
       ||''',to_date('||''''||hiredate||''') );' inserts
  from emp
 where deptno = 10;

INSERTS
---------------------------------------------------------------
insert into emp(empno,ename,hiredate)
values( 7782,'CLARK',to_date('09-JUN-1981 00:00:00') );

insert into emp(empno,ename,hiredate)
values( 7839,'KING',to_date('17-NOV-1981 00:00:00') );

insert into emp(empno,ename,hiredate)
values( 7934,'MILLER',to_date('23-JAN-1982 00:00:00') );
```

3. 讨论

若要创建可移植的脚本（如用于在多个环境下进行测试），用 SQL 生成 SQL 的做法尤其有用。另外，正如上面的例子所示，对于执行批处理维护任务以及一次性找出与多种对象相关的信息，用 SQL 生成 SQL 的做法也非常有用。这种方法简单易学，你练习的次数越多，它就会变得越简单。本例只是希望向你展示一下如何创建属于你自己的动态 SQL 脚本，坦白来说，这并不难做到。你只需要不断尝试就能熟练掌握其中的技巧。

5.7 描述Oracle数据字典视图

1. 问题

你使用的是 Oracle 数据库，但不记得 Oracle 数据库有哪些可用的数据字典视图，也忘了它们包含了哪些列。更糟糕的是，你不方便查找官方文档。

2. 解决方案

本实例仅适用于 Oracle 数据库。Oracle 数据库不仅具有一组内容丰富的数据字典视图，还通过另一组数据字典视图为它们提供详细的注解。这真是精彩的循环。

查询 `DICTIONARY` 视图，并列出数据字典视图及其目的。

```
select table_name, comments
     from dictionary
 order by table_name;

TABLE_NAME                     COMMENTS
------------------------------ --------------------------------------------
ALL_ALL_TABLES                 Description of all object and relational
                               tables accessible to the user

ALL_APPLY                      Details about each apply process that
                               dequeues from the queue visible to the
                               current user
...
```

查询 `DICT_COLUMNS`，并找出某个数据字典视图的各列含义。

```
select column_name, comments
     from dict_columns
 where table_name = 'ALL_TAB_COLUMNS';

COLUMN_NAME                    COMMENTS
------------------------------ ------------------------------------------
OWNER
TABLE_NAME                     Table, view or cluster name
COLUMN_NAME                    Column name
DATA_TYPE                      Datatype of the column
DATA_TYPE_MOD                  Datatype modifier of the column
DATA_TYPE_OWNER                Owner of the datatype of the column
DATA_LENGTH                    Length of the column in bytes
DATA_PRECISION                 Length: decimal digits (NUMBER) or binary
                               digits (FLOAT)
```

3. 讨论

以前，Oracle 数据库的官方文档不像现在这样能通过互联网方便地查看，因而在当时，DICTIONARY 和 DICT_COLUMNS 视图无疑是非常方便的工具。仅仅借助这两个视图，你就能学习到其他全部数据字典视图的知识，进而了解整个数据库。时至今日，了解 DICTIONARY 和 DICT_COLUMNS 视图也非常方便。有时你可能会忘记某个数据库对象对应的数据字典视图名称，你可以用一个含有通配符的查询来找到它。例如，要知道查询哪个视图能得到某个模式下的全部表。

```
select table_name, comments
  from dictionary
 where table_name LIKE '%TABLE%'
 order by table_name;
```

上述查询获取了所有名称中含有 TABLE 一词的数据字典视图。在这里，我们利用了 Oracle 数据库非常一致的数据字典视图命名习惯。描述表的视图名称里往往都带有 TABLE。（有时候会使用 TABLE 的缩写形式 TAB，例如 ALL_TAB_COLUMNS。）

第 6 章

字符串处理

本章主要介绍 SQL 的字符串处理。注意，SQL 并不专门用于处理复杂的字符串。你可能也会发现有时使用 SQL 处理字符串会非常麻烦，令人沮丧。尽管 SQL 存在这些不足之处，各种数据库管理系统仍然提供了许多非常有用的内置函数，我会尽量创造性地利用好它们。本章尤其能反映出我在前言中想要表达的意图：SQL 有好的一面，也有坏的甚至令人厌恶的一面。我希望在你读过本章的内容之后，对于 SQL 在字符串处理方面能做什么和不能做什么能有更加深入的认识。大多数情况下，你会惊喜地发现 SQL 能非常方便地处理许多字符串解析和转换操作，而有时你又可能会惊讶于某些 SQL 特性竟然只是为了某种特定的任务而准备的。

本章的第一个实例非常重要，因为后续的一些实例的解决方案会用到它。大多数情况下，你需要有逐字遍历字符串的能力。但是，使用 SQL 进行这样的操作并不容易。因为 SQL 没有 `Loop` 循环功能（Oracle 的 `MODEL` 子句除外），我们不得不模拟出一种循环操作来实现字符串遍历。我把这种操作称为“遍历字符串”，并在第一个实例里解释了该技巧。这是 SQL 字符串解析处理的基础部分，本章的几乎所有实例都参考或者使用了这一技巧。我强烈建议你先理解它的工作原理。

6.1　遍历字符串

1. 问题

你想遍历一个字符串，并以一个字符一行的形式把它们显示出来，但 SQL 没有 `Loop` 循环功能。例如，你想把 `EMP` 表的 `ENAME` 等于 `KING` 的字符串拆开来显示为 4 行，每行一个字符。

2. 解决方案

使用笛卡儿积生成以每行一个字符的形式来显示字符串所需要的行数。然后，使用数据库内置的字符串解析函数提取我们感兴趣的字符（如果是 SQL Server 的话，要用 SUBSTRING 替换 SUBSTR）。

```
1 select substr(e.ename,iter.pos,1) as C
2   from (select ename from emp where ename = 'KING') e,
3        (select id as pos from t10) iter
4  where iter.pos <= length(e.ename)

C
-
K
I
N
G
```

3. 讨论

要遍历一个字符串里的全部字符，关键在于要先和另一个表做连接查询，该表必须有足够多的行以保证循环操作的次数。本例使用的是 T10 表，该表有 10 行记录（它只有一列，列名为 ID，它的值分别是从 1 到 10）。也就是说，上述查询最多返回 10 行。

下面的例子省略了 ENAME 解析处理，仅展示了 E 和 ITER 的笛卡儿积（例如，某个员工的名字和 T10 表的 10 行数据的笛卡儿积）。

```
select ename, iter.pos
  from (select ename from emp where ename = 'KING') e,
       (select id as pos from t10) iter

ENAME             POS
---------- ----------
KING                1
KING                2
KING                3
KING                4
KING                5
KING                6
KING                7
KING                8
KING                9
KING               10
```

内嵌视图 E 的行数是 1，而内嵌视图 ITER 的行数是 10，所以得到的笛卡儿积就是 10 行。产生这样一个笛卡儿积是使用 SQL 来模拟循环操作的第一步。



把 T10 表作为一个数据透视表是常用技巧。

这个解决方案使用 WHERE 子句在查询语句返回了 4 行数据之后跳出了循环。为了保证结果集的行数等于给定员工名字的字符个数，WHERE 子句把 ITER.POS <= LENGTH(E.ENAME) 作为条件。

```
select ename, iter.pos
  from (select ename from emp where ename = 'KING') e,
       (select id as pos from t10) iter
 where iter.pos <= length(e.ename)

ENAME             POS
---------- ----------
KING                1
KING                2
KING                3
KING                4
```

现在我们得到的记录行数和 E.ENAME 的字符数一样多，接下来可以把 ITER.POS 作为 SUBSTR 的参数，这样就能遍历字符串里的每个字符。ITER.POS 的值会逐行递增，这样每一行都能从 E.ENAME 里提取出一个连续的字符。这就是该解决方案的工作原理。

根据不同的任务目标，我们或许不需要为一个字符串里的每个字符都产生一行数据。下面的查询展示了一个遍历 E.ENAME 的例子，但是查询结果打印的却是字符串的不同部分（不只是单个字符）。

```
select substr(e.ename,iter.pos) a,
       substr(e.ename,length(e.ename)-iter.pos+1) b
  from (select ename from emp where ename = 'KING') e,
       (select id pos from t10) iter
 where iter.pos <= length(e.ename)

A          B
---------- ------
KING       G
ING        NG
NG         ING
G          KING
```

在本章的实例中，最常见的使用场景包括遍历字符串并为其中的每个字符产生一行数据，或者遍历字符串并根据某些特别的字符或分隔符来生成相应行数的记录。

6.2 嵌入引号

1. 问题

你想在字符串常量里嵌入引号，并且希望使用 SQL 产生如下所示的结果。

```
QMARKS
--------------
g'day mate
beavers' teeth
'
```

2. 解决方案

下面的 3 个 SELECT 语句展示了使用引号的不同方式：在一个字符串的中间插入引号和单独使用引号。

```
1 select 'g''day mate' qmarks from t1 union all
2 select 'beavers'' teeth'    from t1 union all
3 select ''''                 from t1
```

3. 讨论

讨论引号的处理时，把它和圆括号做类比通常会更容易理解。我们如果写下左括号，也必须接着写下右括号。引号也是一样。要记住，在任何字符串里引号的个数都应该是一个偶数。要想在字符串中间插入引号，需要使用两个引号才行。

```
select 'apples core', 'apple''s core',
        case when '' is null then 0 else 1 end
  from t1

 'APPLESCORE  'APPLE''SCOR  CASEWHEN''ISNULLTHEN0ELSE1END
 -----------  -----------  -----------------------------
 apples core  apple's core                            0
```

下面的查询用于揭示引号的真正作用。两个外层的引号用于定义一个字符串常量，在该字符串常量里还使用了两个引号来代表一个引号，我们实际上希望它显示的内容就是这一个引号。

```
select '''' as quote from t1

 Q
 -
 '
```

处理引号时，还要记住如果一个字符串里只包含两个引号，并且这两个引号中间没有任何字符，那么这个字符串是 Null。

6.3　统计字符出现的次数

1. 问题

你想统计某个字符或者子字符串在给定字符串里出现的次数，考虑如下的字符串。

```
10,CLARK,MANAGER
```

你想知道该字符串里有多少个逗号。

2. 解决方案

字符串的总长度减去去掉逗号之后的字符串长度，就得到了逗号的个数。所有的数据库管理系统都提供了获取字符串长度的函数以及从字符串里删除字符的函数。大多数情况下，这些函数分别是 LENGTH 和 REPLACE（SQL Server 用户需要用内置函数 LEN 替换 LENGTH）。

```
1 select (length('10,CLARK,MANAGER')-
2        length(replace('10,CLARK,MANAGER',',','')))/length(',')
3        as cnt
4   from t1
```

3. 讨论

使用简单的减法运算就可以解决这个问题。第 1 行调用 LENGTH 函数获取字符串总长

度，第 2 行仍然调用 LENGTH 函数获取不含逗号的字符串长度，而逗号的删除则借助了 REPLACE 函数。

把上述两个长度相减，得到的差值就是字符串里逗号的个数。最后的除法运算是用上述两个长度的差值除以我们正在搜索的那个字符串长度。如果被搜索的字符串的长度大于 1 的话，就必须使用除法运算。下面的例子统计在字符串 HELLO HELLO 中出现了多少个 LL，如果没有进行除法运算的话，就不会得到正确的结果。

```
select
       (length('HELLO HELLO')-
       length(replace('HELLO HELLO','LL','')))/length('LL')
       as correct_cnt,
       (length('HELLO HELLO')-
       length(replace('HELLO HELLO','LL',''))) as incorrect_cnt
  from t1

CORRECT_CNT INCORRECT_CNT
----------- -------------
          2             4
```

6.4 删除不想要的字符

1. 问题

你想从你的数据里删除指定的字符，考虑下面的结果集。

```
ENAME             SAL
---------- ----------
SMITH             800
ALLEN            1600
WARD             1250
JONES            2975
MARTIN           1250
BLAKE            2850
CLARK            2450
SCOTT            3000
KING             5000
TURNER           1500
ADAMS            1100
JAMES             950
FORD             3000
MILLER           1300
```

你希望从上面的数据里删除所有的 0 和元音字母，并将删除后的值显示在 STRIPPED1 列和 STRIPPED2 列中。

```
ENAME      STRIPPED1         SAL STRIPPED2
---------- ---------- ---------- ---------
SMITH      SMTH              800 8
ALLEN      LLN              1600 16
WARD       WRD              1250 125
JONES      JNS              2975 2975
MARTIN     MRTN             1250 125
```

```
BLAKE      BLK              2850 285
CLARK      CLRK             2450 245
SCOTT      SCTT             3000 3
KING       KNG              5000 5
TURNER     TRNR             1500 15
ADAMS      DMS              1100 11
JAMES      JMS               950 95
FORD       FRD              3000 3
MILLER     MLLR             1300 13
```

2. 解决方案

每个数据库管理系统都提供了可以删除一个字符串里不想要的字符的函数，其中最有用的函数是 REPLACE 和 TRANSLATE。

DB2

使用内置函数 TRANSLATE 和 REPLACE 删除不想要的字符和字符串。

```
1 select ename,
2        replace(translate(ename,'aaaaa','AEIOU'),'a','') stripped1,
3        sal,
4        replace(cast(sal as char(4)),'0','') stripped2
5   from emp
```

MySQL 和 SQL Server

MySQL 和 SQL Server 没有提供 TRANSLATE 函数，因而需要多次调用 REPLACE 函数。

```
1  select ename,
2         replace(
3         replace(
4         replace(
5         replace(
6         replace(ename,'A',''),'E',''),'I',''),'O',''),'U','')
7         as stripped1,
8         sal,
9         replace(sal,0,'') stripped2
10   from emp
```

Oracle 和 PostgreSQL

使用内置函数 TRANSLATE 和 REPLACE 删除不想要的字符和字符串。

```
1  select ename,
2         replace(translate(ename,'AEIOU','aaaaa'),'a')
3         as stripped1,
4         sal,
5         replace(sal,0,'') as stripped2
6    from emp
```

3. 讨论

内置函数 REPLACE 会删除 0。为了删除元音字母，先使用 TRANSLATE 函数把元音字母替换成一个特殊的字符（我选择了字母 a，当然也可以选其他字符），然后使用 REPLACE 函数删除这个特殊字符。

6.5 分离数字和字符数据

1. 问题

你很不幸地把数字和字符数据混合存放进一列。你想把其中的数字数据和字符数据分开，考虑如下的结果集。

```
DATA
---------------
SMITH800
ALLEN1600
WARD1250
JONES2975
MARTIN1250
BLAKE2850
CLARK2450
SCOTT3000
KING5000
TURNER1500
ADAMS1100
JAMES950
FORD3000
MILLER1300
```

你希望得到如下的结果集。

```
ENAME             SAL
---------- ----------
SMITH             800
ALLEN            1600
WARD             1250
JONES            2975
MARTIN           1250
BLAKE            2850
CLARK            2450
SCOTT            3000
KING             5000
TURNER           1500
ADAMS            1100
JAMES             950
FORD             3000
MILLER           1300
```

2. 解决方案

使用内置函数 TRANSLATE 和 REPLACE 来分离字符数据和数字数据。与本章的其他实例类似，此处的技巧在于使用 TRANSLATE 函数把多种字符替换成一个指定的字符。这样一来只要用一个数字就能代表所有数字，一个字符就能代表所有字符，因此我们就不再需要逐一查找多个数字或字符了。

DB2

使用 TRANSLATE 和 REPLACE 函数分离数字和字符数据。

```
1 select replace(
2      translate(data,'0000000000','0123456789'),'0','') ename,
3           cast(
4        replace(
5      translate(lower(data),repeat('z',26),
6            'abcdefghijklmnopqrstuvwxyz'),'z','') as integer) sal
7     from (
8   select ename||cast(sal as char(4)) data
9     from emp
10           ) x
```

Oracle

使用 TRANSLATE 和 REPLACE 函数分离数字和字符数据。

```
 1  select replace(
 2       translate(data,'0123456789','0000000000'),'0') ename,
 3       to_number(
 4         replace(
 5         translate(lower(data),
 6                   'abcdefghijklmnopqrstuvwxyz',
 7                    rpad('z',26,'z')),'z')) sal
 8    from (
 9  select ename||sal data
10    from emp
11         )
```

PostgreSQL

使用 TRANSLATE 和 REPLACE 函数分离数字和字符数据。

```
 1 select replace(
 2      translate(data,'0123456789','0000000000'),'0','') as ename,
 3           cast(
 4        replace(
 5      translate(lower(data),
 6                'abcdefghijklmnopqrstuvwxyz',
 7                rpad('z',26,'z')),'z','') as integer) as sal
 8   from (
 9 select ename||sal as data
10   from emp
11        ) x
```

3. 讨论

每个数据库管理系统的语法都略有不同，但方法是一样的。在本节的讨论中，我将以 Oracle 解决方案为主。解决本问题的关键在于分离数字和字符数据，使用 TRANSLATE 和 REPLACE 函数可以实现这一点。为了提取数字，首先用 TRANSLATE 函数把所有的字符数据分离出来。

```
select data,
       translate(lower(data),
                'abcdefghijklmnopqrstuvwxyz',
                 rpad('z',26,'z')) sal
  from (select ename||sal data from emp)
```

```
DATA                 SAL
-------------------- -------------------
SMITH800             zzzzz800
ALLEN1600            zzzzz1600
WARD1250             zzzz1250
JONES2975            zzzzz2975
MARTIN1250           zzzzzz1250
BLAKE2850            zzzzz2850
CLARK2450            zzzzz2450
SCOTT3000            zzzzz3000
KING5000             zzzz5000
TURNER1500           zzzzzz1500
ADAMS1100            zzzzz1100
JAMES950             zzzzz950
FORD3000             zzzz3000
MILLER1300           zzzzzz1300
```

使用 TRANSLATE 函数把每一个非数字字符都替换为小写字母 z。然后使用 REPLACE 函数删除所有的小写字母 z，这样就只留下数字字符，我们可以将其转换为一个数字。

```
select data,
       to_number(
         replace(
       translate(lower(data),
                 'abcdefghijklmnopqrstuvwxyz',
                  rpad('z',26,'z')),'z')) sal
  from (select ename||sal data from emp)

DATA                        SAL
-------------------- ----------
SMITH800                    800
ALLEN1600                  1600
WARD1250                   1250
JONES2975                  2975
MARTIN1250                 1250
BLAKE2850                  2850
CLARK2450                  2450
SCOTT3000                  3000
KING5000                   5000
TURNER1500                 1500
ADAMS1100                  1100
JAMES950                    950
FORD3000                   3000
MILLER1300                 1300
```

为了提取非数字字符，需要使用 TRANSLATE 函数隔离数字字符。

```
select data,
       translate(data,'0123456789','0000000000') ename
  from (select ename||sal data from emp)

DATA                 ENAME
-------------------- ----------
SMITH800             SMITH000
ALLEN1600            ALLEN0000
```

```
WARD1250             WARD0000
JONES2975            JONES0000
MARTIN1250           MARTIN0000
BLAKE2850            BLAKE0000
CLARK2450            CLARK0000
SCOTT3000            SCOTT0000
KING5000             KING0000
TURNER1500           TURNER0000
ADAMS1100            ADAMS0000
JAMES950             JAMES000
FORD3000             FORD0000
MILLER1300           MILLER0000
```

使用 TRANSLATE 函数把每一个数字字符替换为 0，然后使用 REPLACE 函数删除每条记录中出现的 0，剩下的就只有非数字字符。

```
select data,
       replace(translate(data,'0123456789','0000000000'),'0') ename
  from (select ename||sal data from emp)

DATA                 ENAME
-------------------- --------
SMITH800             SMITH
ALLEN1600            ALLEN
WARD1250             WARD
JONES2975            JONES
MARTIN1250           MARTIN
BLAKE2850            BLAKE
CLARK2450            CLARK
SCOTT3000            SCOTT
KING5000             KING
TURNER1500           TURNER
ADAMS1100            ADAMS
JAMES950             JAMES
FORD3000             FORD
MILLER1300           MILLER
```

最后，将上述两个方法结合起来就是本问题的解决方案。

6.6 判断含有字母和数字的字符串

1. 问题

你想从一个表里筛选出部分行数据，筛选条件是你感兴趣的那个列只包含字母和数字字符，考虑下面的视图 V（SQL Server 用户需要把字符串连接操作符 || 替换为 +）。

```
create view V as
select ename as data
  from emp
 where deptno=10
 union all
select ename||', $'|| cast(sal as char(4)) ||'.00' as data
  from emp
 where deptno=20
```

```
 union all
select ename|| cast(deptno as char(4)) as data
  from emp
 where deptno=30
```

视图 V 代表了你要查询的表，它包含如下所示的数据。

```
DATA
--------------------
CLARK
KING
MILLER
SMITH, $800.00
JONES, $2975.00
SCOTT, $3000.00
ADAMS, $1100.00
FORD, $3000.00
ALLEN30
WARD30
MARTIN30
BLAKE30
TURNER30
JAMES30
```

然而，你只希望从视图 V 中提取出如下所示的记录。

```
DATA
-------------
CLARK
KING
MILLER
ALLEN30
WARD30
MARTIN30
BLAKE30
TURNER30
JAMES30
```

总之，你想过滤掉那些除了字母和数字还包含其他字符的行。

2. 解决方案

首先找出字符串中所有可能出现的非字母数字字符，这似乎是更为直观的解决思路。但恰恰与之相反，我们发现从反面着手更容易：首先找出所有的字母字符和数字字符。如此一来，先把所有的字母字符和数字字符转换成一个单一的字符，然后就能把它们当作一个字符。这么做的好处是，经过转换处理之后这些字母和数字可以被当作一个整体来操作。一旦生成了原有字符串的副本，并把其中的字母字符和数字字符替换成某个指定的字符，很容易就可以将字母字符和数字字符从其他字符中分离出来。

DB2

使用 TRANSLATE 函数将字母字符和数字字符都替换成单一字符，然后找出那些除了该字符还包含其他字符的行。对于 DB2 用户来说，需要在视图 V 中调用 CAST 函数。否则，会因为数据类型转换错误而导致视图创建失败。转换为 CHAR 类型时尤其要注意，因为 CHAR 的

长度是固定的（长度不足的部分会被填充上）。

```
1 select data
2   from V
3  where translate(lower(data),
4                  repeat('a',36),
5                  '0123456789abcdefghijklmnopqrstuvwxyz') =
6                  repeat('a',length(data))
```

MySQL

在 MySQL 中，视图 V 的语法稍有不同。

```
create view V as
select ename as data
  from emp
 where deptno=10
 union all
select concat(ename,', $',sal,'.00') as data
  from emp
 where deptno=20
 union all
select concat(ename,deptno) as data
  from emp
 where deptno=30
```

使用正则表达式能方便地找出包含非字母数字字符的行。

```
1 select data
2   from V
3  where data regexp '[^0-9a-zA-Z]' = 0
```

Oracle 和 PostgreSQL

使用 TRANSLATE 函数把字母字符和数字字符替换成单一字符，然后找出那些除了该字符还包含其他字符的行。对于 Oracle 和 PostgreSQL 而言，视图 V 不需要调用 CAST 函数。转换为 CHAR 类型时尤其要注意，因为 CHAR 的长度是固定的（长度不足的部分会被填充上）。如果确实需要转换类型，那么就转成 VARCHAR 或 VARCHAR2 类型。

```
1 select data
2   from V
3  where translate(lower(data),
4                  '0123456789abcdefghijklmnopqrstuvwxyz',
5                  rpad('a',36,'a')) = rpad('a',length(data),'a')
```

SQL Server

因为 SQL Server 不支持 TRANSLATE 函数，我们必须遍历每一行数据，并找出那些包含非字母数字字符的行。有很多种办法可以实现这一点，下面的解决方案的思路是评估每个字符的 ASCII 值。

```
 1  select data
 2    from (
 3  select v.data, iter.pos,
 4         substring(v.data,iter.pos,1) c,
 5         ascii(substring(v.data,iter.pos,1)) val
```

```
 6    from v,
 7         ( select id as pos from t100 ) iter
 8   where iter.pos <= len(v.data)
 9         ) x
10   group by data
11   having min(val) between 48 and 122
```

3. 讨论

上述解决方案的关键在于能同时查看多个字符。通过使用 `TRANSLATE` 函数，我们可以很容易处理全部数字或全部字符，而且不需要循环枚举并逐一查看每个字符。

DB2、Oracle 和 PostgreSQL

视图 `V` 的 14 行数据里只有 9 行是字母字符和数字字符。为了筛选出只包含字母字符和数字字符的行，直接使用 TRANSLATE 函数即可。在本例中，`TRANSLATE` 函数把字符 `0` ～ `9` 和 `a` ～ `z` 都转换成了 `a`。一旦完成了这一转换，下一步就要比较转换后的行数据和一个（与当前行的数据）具有相同长度并且只包括 `a` 的字符串。如果二者相同，那么我们就可以认定该字符串仅由字母和数字构成，而且不含其他字符。

使用 `TRANSLATE` 函数（这里以 Oracle 语法为例）。

```
where translate(lower(data),
                '0123456789abcdefghijklmnopqrstuvwxyz',
                rpad('a',36,'a'))
```

我们把全部数字和字母字符都替换成了一个独特的字符（我这里选择了 `a`）。一旦这种替换完成，那些仅由字母和数字组成的字符串就变成了一个由单一字符（本例中是 `a`）构成的字符串。这一点可以通过单独执行 `TRANSLATE` 函数来进行验证。

```
select data, translate(lower(data),
                  '0123456789abcdefghijklmnopqrstuvwxyz',
                  rpad('a',36,'a'))
  from V

DATA                 TRANSLATE(LOWER(DATA)
-------------------- ---------------------
CLARK                aaaaa
...
SMITH, $800.00       aaaaa, $aaa.aa
...
ALLEN30              aaaaaaa
...
```

虽然字母字符和数字字符被替换掉了，但字符串的长度并没有发生变化。由于长度是一样的，被筛选出来的行就是那些调用了 `TRANSLATE` 函数之后返回值里只包括 `a` 的行。通过比较原字符串的长度和只包含 `a` 的字符串长度，我们保留了相等的行，过滤掉了其他的行。

```
select data, translate(lower(data),
                  '0123456789abcdefghijklmnopqrstuvwxyz',
                  rpad('a',36,'a')) translated,
        rpad('a',length(data),'a') fixed
```

```
  from V

DATA                 TRANSLATED           FIXED
-------------------- -------------------- ----------------
CLARK                aaaaa                aaaaa
...
SMITH, $800.00       aaaaa, $aaa.aa       aaaaaaaaaaaaaa
...
ALLEN30              aaaaaaa              aaaaaaa
...
```

最后一步就是只保留那些 TRANSLATED 和 FIXED 相等的字符串。

MySQL

WHERE 子句里的表达式如下所示：

```
where data regexp '[^0-9a-zA-Z]' = 0
```

上述条件使得那些仅包含数字和字母的行会被筛选出来。方括号里的取值范围 0-9a-zA-Z 表示所有可能出现的数字和字母。符号 ^ 表示否定，因而该表达式可被解释为“非数字或非字母”。返回值等于 1 代表 TRUE，0 代表 FALSE，因此整个表达式的意思是：“执行非数字和字母字符匹配操作，并返回结果等于 FALSE 的行。”

SQL Server

首先遍历视图 V 的每一行数据，DATA 列的每一个字符都会被作为一行返回。C 列的值代表了构成 DATA 值的每一个字符。

```
+-----------------+------+------+------+
| data            | pos  | c    | val  |
+-----------------+------+------+------+
| ADAMS, $1100.00 |    1 | A    |   65 |
| ADAMS, $1100.00 |    2 | D    |   68 |
| ADAMS, $1100.00 |    3 | A    |   65 |
| ADAMS, $1100.00 |    4 | M    |   77 |
| ADAMS, $1100.00 |    5 | S    |   83 |
| ADAMS, $1100.00 |    6 | ,    |   44 |
| ADAMS, $1100.00 |    7 |      |   32 |
| ADAMS, $1100.00 |    8 | $    |   36 |
| ADAMS, $1100.00 |    9 | 1    |   49 |
| ADAMS, $1100.00 |   10 | 1    |   49 |
| ADAMS, $1100.00 |   11 | 0    |   48 |
| ADAMS, $1100.00 |   12 | 0    |   48 |
| ADAMS, $1100.00 |   13 | .    |   46 |
| ADAMS, $1100.00 |   14 | 0    |   48 |
| ADAMS, $1100.00 |   15 | 0    |   48 |
```

内嵌视图 X 不仅会逐行返回 DATA 列的每一个字符，还会提供每个字符的 ASCII 值。这是 SQL Server 的专有功能，ASCII 取值范围 48 ～ 122 代表了字母字符和数字字符。理解了这一点，我们就可以对 DATA 进行分组，并过滤掉 ASCII 值不在 48 ～ 122 范围内的值。

6.7　提取姓名的首字母

1. 问题

你想把姓名变成首字母的形式，考虑人名 Stewie Griffin，你希望得到 S.G.。

2. 解决方案

注意，SQL 的灵活性比不上 C 语言或 Python 这样的编程语言。因此，很难使用 SQL 创建一个处理姓名格式转换的通用解决方案。下面给出的解决方案仅适用于两种格式：要么是 First Name 和 Last Name 的组合，要么是 First Name、Middle Name（全称或者首字母均可）和 Last Name 的组合。

DB2

使用内置函数 REPLACE、TRANSLATE 和 REPEAT 提取首字母。

```
1 select replace(
2        replace(
3        translate(replace('Stewie Griffin', '.', ''),
4                  repeat('#',26),
5                  'abcdefghijklmnopqrstuvwxyz'),
6                   '#','' ), ' ','.' )
7                  ||'.'
8   from t1
```

MySQL

使用内置函数 CONCAT、CONCAT_WS、SUBSTRING 和 SUBSTRING_INDEX 提取首字母。

```
1  select case
2           when cnt = 2 then
3             trim(trailing '.' from
4                  concat_ws('.',
5                   substr(substring_index(name,' ',1),1,1),
6                   substr(name,
7                          length(substring_index(name,' ',1))+2,1),
8                   substr(substring_index(name,' ',-1),1,1),
9                   '.'))
10          else
11            trim(trailing '.' from
12                 concat_ws('.',
13                  substr(substring_index(name,' ',1),1,1),
14                  substr(substring_index(name,' ',-1),1,1)
15                  ))
16          end as initials
17   from (
18 select name,length(name)-length(replace(name,' ','')) as cnt
19   from (
20 select replace('Stewie Griffin','.','') as name from t1
21         )y
22         )x
```

Oracle 和 PostgreSQL

使用内置函数 REPLACE、TRANSLATE 和 RPAD 提取首字母。

```
1 select replace(
2        replace(
3        translate(replace('Stewie Griffin', '.', ''),
4                   'abcdefghijklmnopqrstuvwxyz',
5                   rpad('#',26,'#') ), '#','' ),' ','.' ) ||'.'
6   from t1
```

SQL Server

在写作本书时，SQL Server 尚不支持 TRANSLATE 函数和 CONCAT_WS 函数。

3.讨论

通过分离出大写字母，我们就能从姓名中提取首字母。下面详细解释针对各种数据库的解决方案。

DB2

REPLACE 函数会删除姓名里出现的英文句号（因为有时候 Middle Name 会以首字母形式表示），而 TRANSLATE 函数会把非大写字母都替换为字符 #。

```
select translate(replace('Stewie Griffin', '.', ''),
                 repeat('#',26),
                 'abcdefghijklmnopqrstuvwxyz')
  from t1

TRANSLATE('STE
--------------
S##### G######
```

此时，除了首字母外，名字的其他部分都变成了 #。然后使用 REPLACE 函数删除所有的 #。

```
select replace(
       translate(replace('Stewie Griffin', '.', ''),
                  repeat('#',26),
                  'abcdefghijklmnopqrstuvwxyz'),'#','')
  from t1

REP
---
S G
```

再次使用 REPLACE 函数把空格替换为英文句号。

```
select replace(
       replace(
       translate(replace('Stewie Griffin', '.', ''),
                  repeat('#',26),
                 'abcdefghijklmnopqrstuvwxyz'),'#',''),' ','.') || '.'
  from t1

REPLA
-----
S.G
```

最后，在姓名首字母的末尾添加英文句号。

Oracle 和 PostgreSQL

REPLACE 函数会删除姓名里出现的英文句号（因为有时候 Middle Name 会以首字母形式表示），而 TRANSLATE 函数会把非大写字母都替换为字符 #。

```
select translate(replace('Stewie Griffin','.',''),
                 'abcdefghijklmnopqrstuvwxyz',
                 rpad('#',26,'#'))
  from t1

TRANSLATE('STE
--------------
S##### G######
```

此时，除了首字母外，姓名的其他部分都变成了 #。然后使用 REPLACE 函数删除掉所有的 #。

```
select replace(
       translate(replace('Stewie Griffin','.',''),
                 'abcdefghijklmnopqrstuvwxyz',
                  rpad('#',26,'#')),'#','')
  from t1

REP
---
S G
```

再次使用 REPLACE 函数把空格替换为英文句号。

```
select replace(
       replace(
     translate(replace('Stewie Griffin','.',''),
               'abcdefghijklmnopqrstuvwxyz',
                rpad('#',26,'#') ),'#',''),' ','.') || '.'
  from t1

REPLA
-----
S.G
```

最后，在姓名首字母的末尾添加英文句号。

MySQL

内嵌视图 Y 用于删除姓名中出现的英文句号。内嵌视图 X 可以找出姓名中空格符的个数，以便调用适当次数的 SUBSTR 函数来提取首字母。先后三次调用 SUBSTRING_INDEX 函数，根据空格的位置把字符串拆成三个单独的部分。本例中出现的姓名只包括 First Name 和 Last Name，CASE 语句的 ELSE 部分代码会被执行。

```
select substr(substring_index(name, ' ',1),1,1) as a,
       substr(substring_index(name,' ',-1),1,1) as b
  from (select 'Stewie Griffin' as name from t1) x

A B
- -
S G
```

如果问题中的姓名包含 Middle Name 或其首字母，那么执行下面的代码可以得到首字母。

```
substr(name,length(substring_index(name, ' ',1))+2,1)
```

上面的查询先找出 First Name 的结束位置，并前进两个字符位置移动到 Middle Name 或其首字母的开始位置；计算结果将作为 SUBSTR 函数的开始位置。因为只需要保留第一个字符，所以 Middle Name 或其首字母能被成功地返回。然后，提取出的首字母会传递给 CONCAT_WS 函数，这样就能用英文句号分割各个首字母。

```
select concat_ws('.',
                 substr(substring_index(name, ' ',1),1,1),
                 substr(substring_index(name,' ',-1),1,1),
                 '.' ) a
  from (select 'Stewie Griffin' as name from t1) x

A
-----
S.G..
```

最后，删除首字母中无关的英文句号。

6.8 按照子字符串排序

1. 问题

你想根据一个子字符串对结果集进行排序，考虑下面的记录。

```
ENAME
----------
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
ADAMS
JAMES
FORD
MILLER
```

你希望按照每个名字的最后两个字符对上述记录进行排序。

```
ENAME
---------
ALLEN
TURNER
MILLER
JONES
JAMES
MARTIN
```

```
BLAKE
ADAMS
KING
WARD
FORD
CLARK
SMITH
SCOTT
```

2. 解决方案

解决这个问题的关键在于使用数据库管理系统的内置函数提取出用作排序标准的子字符串，通常使用 SUBSTR 函数来实现这一点。

DB2、Oracle、MySQL 和 PostgreSQL

使用内置函数 LENGTH 和 SUBSTR，根据字符串的特定部分排序。

```
1 select ename
2   from emp
3  order by substr(ename,length(ename)-1,2)
```

SQL Server

使用 SUBSTRING 函数和 LEN 函数，根据子字符串的特定部分排序。

```
1 select ename
2   from emp
3  order by substring(ename,len(ename)-1,2)
```

3. 讨论

通过在 ORDER BY 子句里使用 SUBSTR 表达式，我们可以选择一个字符串的任意部分用于结果集的排序。其实，也可以不使用 SUBSTR 函数，我们可以基于任何表达式对结果集排序。

6.9 根据字符串里的数字排序

1. 问题

你希望根据字符串里的数字对结果集进行排序，考虑下面的视图。

```
create view V as
select e.ename ||' '||
       cast(e.empno as char(4))||' '||
       d.dname as data
  from emp e, dept d
 where e.deptno=d.deptno
```

下面是上述视图返回的数据。

```
DATA
-----------------------------
CLARK   7782 ACCOUNTING
KING    7839 ACCOUNTING
MILLER  7934 ACCOUNTING
SMITH   7369 RESEARCH
JONES   7566 RESEARCH
```

```
SCOTT   7788 RESEARCH
ADAMS   7876 RESEARCH
FORD    7902 RESEARCH
ALLEN   7499 SALES
WARD    7521 SALES
MARTIN  7654 SALES
BLAKE   7698 SALES
TURNER  7844 SALES
JAMES   7900 SALES
```

以上数据包括员工名字、员工编号和部门名称三部分数据，你希望按照中间的员工编号对该数据进行排序。

```
DATA
---------------------------
SMITH   7369 RESEARCH
ALLEN   7499 SALES
WARD    7521 SALES
JONES   7566 RESEARCH
MARTIN  7654 SALES
BLAKE   7698 SALES
CLARK   7782 ACCOUNTING
SCOTT   7788 RESEARCH
KING    7839 ACCOUNTING
TURNER  7844 SALES
ADAMS   7876 RESEARCH
JAMES   7900 SALES
FORD    7902 RESEARCH
MILLER  7934 ACCOUNTING
```

2. 解决方案

下面的每一种解决方案都使用了各个数据库特有的函数和语法，但方法（利用内置函数 REPLACE 和 TRANSLATE）却是相同的。基本思路都是使用 REPLACE 和 TRANSLATE 函数删除字符串里的非数字字符，只留下用于排序的数字。

DB2

使用内置函数 REPLACE 和 TRANSLATE 提取字符串里的数字，并按照数字排序。

```
1 select data
2   from V
3  order by
4         cast(
5      replace(
6    translate(data,repeat('#',length(data)),
7      replace(
8    translate(data,'##########','0123456789'),
9              '#','')),'#','') as integer)
```

Oracle

使用内置函数 REPLACE 和 TRANSLATE 提取字符串里的数字，并按照数字排序。

```
1 select data
2   from V
```

```
3  order by
4         to_number(
5           replace(
6         translate(data,
7           replace(
8         translate(data,'0123456789','##########'),
9                  '#'),rpad('#',20,'#')),'#'))
```

PostgreSQL

使用内置函数 REPLACE 和 TRANSLATE 提取字符串里的数字，并按照数字排序。

```
1 select data
2   from V
3  order by
4         cast(
5      replace(
6    translate(data,
7      replace(
8    translate(data,'0123456789','##########'),
9             '#',''),rpad('#',20,'#')),'#','') as integer)
```

MySQL 和 SQL Server

在写作本书时，这两种数据库尚不支持 TRANSLATE 函数。

3. 讨论

视图 V 只是为了提供演示本实例解决方案的数据。该视图只是简单地把 EMP 表的一些列串联起来。上述解决方案展示了如何以串联后的文本作为输入数据，并按照嵌入其中的员工编号进行排序。

各个解决方案的 ORDER BY 子句虽然看起来有点吓人，但效果不错。如果我们一段一段地仔细阅读的话，就会发现它其实不难理解。为了按照字符串里的数字排序，最简单的办法就是删除所有的非数字字符。删除非数字字符后，把数字字符变成数值类型，并进行排序。在开始解释每一次函数调用之前，我们要先理解各个函数被调用的次序。先从最内层的 TRANSLATE 函数调用（每个解决方案的第 8 行）开始，我们可以看到：

(1) TRANSLATE 函数（第 8 行）被调用，把执行结果传递给；
(2) REPLACE 函数（第 7 行），并把执行结果传递给；
(3) TRANSLATE 函数（第 6 行），并把执行结果传递给；
(4) REPLACE 函数（第 5 行），其执行结果被返回，最后；
(5) 转换为数值类型。

第一步是把数字替换为一个特别的字符，它和去掉数字后字符串里剩下的字符都不相同。本例中我选择了 #，并使用 TRANSLATE 函数把所有的数字都替换成 #。例如，下面的查询左边显示的是原来的字符串，右边显示的是第一次转换后得到的结果。

```
select data,
       translate(data,'0123456789','##########') as tmp
  from V
```

```
DATA                           TMP
------------------------------ -----------------------
CLARK   7782 ACCOUNTING        CLARK   #### ACCOUNTING
KING    7839 ACCOUNTING        KING    #### ACCOUNTING
MILLER  7934 ACCOUNTING        MILLER  #### ACCOUNTING
SMITH   7369 RESEARCH          SMITH   #### RESEARCH
JONES   7566 RESEARCH          JONES   #### RESEARCH
SCOTT   7788 RESEARCH          SCOTT   #### RESEARCH
ADAMS   7876 RESEARCH          ADAMS   #### RESEARCH
FORD    7902 RESEARCH          FORD    #### RESEARCH
ALLEN   7499 SALES             ALLEN   #### SALES
WARD    7521 SALES             WARD    #### SALES
MARTIN  7654 SALES             MARTIN  #### SALES
BLAKE   7698 SALES             BLAKE   #### SALES
TURNER  7844 SALES             TURNER  #### SALES
JAMES   7900 SALES             JAMES   #### SALES
```

TRANSLATE 函数找到每个字符串里的数字字符，并逐一替换为 #。转换后字符串被传递到 REPLACE 函数（第 7 行），它会删除所有的 #。

```
select data,
replace(
translate(data,'0123456789','##########'),'#') as tmp
  from V

DATA                           TMP
------------------------------ --------------------
CLARK   7782 ACCOUNTING        CLARK    ACCOUNTING
KING    7839 ACCOUNTING        KING     ACCOUNTING
MILLER  7934 ACCOUNTING        MILLER   ACCOUNTING
SMITH   7369 RESEARCH          SMITH    RESEARCH
JONES   7566 RESEARCH          JONES    RESEARCH
SCOTT   7788 RESEARCH          SCOTT    RESEARCH
ADAMS   7876 RESEARCH          ADAMS    RESEARCH
FORD    7902 RESEARCH          FORD     RESEARCH
ALLEN   7499 SALES             ALLEN    SALES
WARD    7521 SALES             WARD     SALES
MARTIN  7654 SALES             MARTIN   SALES
BLAKE   7698 SALES             BLAKE    SALES
TURNER  7844 SALES             TURNER   SALES
JAMES   7900 SALES             JAMES    SALES
```

然后，上述结果再一次被传递给 TRANSLATE 函数，但这次是本解决方案第二次（最外层）调用 TRANSLATE 函数。该 TRANSLATE 函数在原来的字符串中搜索和 TMP 相匹配的字符。如果找到的话，就把它们都替换成 #。这一转换使得所有非数字字符能够被当作单一字符来处理（因为它们都被替换成了相同的字符）。

```
select data, translate(data,
             replace(
             translate(data,'0123456789','##########'),
                       '#'),
                       rpad('#',length(data),'#')) as tmp
  from V
```

```
DATA                           TMP
------------------------------ ---------------------------
CLARK   7782 ACCOUNTING        #######7782###########
KING    7839 ACCOUNTING        #######7839###########
MILLER  7934 ACCOUNTING        #######7934###########
SMITH   7369 RESEARCH          #######7369#########
JONES   7566 RESEARCH          #######7566#########
SCOTT   7788 RESEARCH          #######7788#########
ADAMS   7876 RESEARCH          #######7876#########
FORD    7902 RESEARCH          #######7902#########
ALLEN   7499 SALES             #######7499######
WARD    7521 SALES             #######7521######
MARTIN  7654 SALES             #######7654######
BLAKE   7698 SALES             #######7698######
TURNER  7844 SALES             #######7844######
JAMES   7900 SALES             #######7900######
```

接下来，通过调用 REPLACE 函数（第 5 行）删除所有的 #，只留下数字字符。

```
select data, replace(
             translate(data,
             replace(
           translate(data,'0123456789','##########'),
                     '#'),
                     rpad('#',length(data),'#')),'#') as tmp
  from V

DATA                           TMP
------------------------------ -----------
CLARK   7782 ACCOUNTING        7782
KING    7839 ACCOUNTING        7839
MILLER  7934 ACCOUNTING        7934
SMITH   7369 RESEARCH          7369
JONES   7566 RESEARCH          7566
SCOTT   7788 RESEARCH          7788
ADAMS   7876 RESEARCH          7876
FORD    7902 RESEARCH          7902
ALLEN   7499 SALES             7499
WARD    7521 SALES             7521
MARTIN  7654 SALES             7654
BLAKE   7698 SALES             7698
TURNER  7844 SALES             7844
JAMES   7900 SALES             7900
```

最后，使用数据库管理系统中合适的函数（通常是 CAST）把 TMP 转换为数值类型（第 4 行），结果如下所示。

```
select data, to_number(
              replace(
             translate(data,
             replace(
       translate(data,'0123456789','##########'),
                     '#'),
                     rpad('#',length(data),'#')),'#')) as tmp
  from V
```

```
DATA                           TMP
------------------------------ ----------
CLARK   7782 ACCOUNTING        7782
KING    7839 ACCOUNTING        7839
MILLER  7934 ACCOUNTING        7934
SMITH   7369 RESEARCH          7369
JONES   7566 RESEARCH          7566
SCOTT   7788 RESEARCH          7788
ADAMS   7876 RESEARCH          7876
FORD    7902 RESEARCH          7902
ALLEN   7499 SALES             7499
WARD    7521 SALES             7521
MARTIN  7654 SALES             7654
BLAKE   7698 SALES             7698
TURNER  7844 SALES             7844
JAMES   7900 SALES             7900
```

当编写类似这样的查询语句时，不妨把写好的表达式放入 SELECT 列表里试着执行一下，这会非常有用。因为我们能很容易看到中间结果，直到得出最终的解决方案。然而，因为本实例的重点是对结果集进行排序，所以最终仍需要把所有的函数调用都放进 ORDER BY 子句里。

```
select data
  from V
 order by
        to_number(
          replace(
        translate( data,
          replace(
        translate( data,'0123456789','##########'),
                  '#'),rpad('#',length(data),'#')),'#'))

DATA
---------------------------
SMITH   7369 RESEARCH
ALLEN   7499 SALES
WARD    7521 SALES
JONES   7566 RESEARCH
MARTIN  7654 SALES
BLAKE   7698 SALES
CLARK   7782 ACCOUNTING
SCOTT   7788 RESEARCH
KING    7839 ACCOUNTING
TURNER  7844 SALES
ADAMS   7876 RESEARCH
JAMES   7900 SALES
FORD    7902 RESEARCH
MILLER  7934 ACCOUNTING
```

最后值得注意的是，本例的视图数据包含 3 个字段，其中只有一个字段是数字。如果有多个数字字段，就需要将它们拼接成一个数字，然后再排序。

6.10 创建分隔列表

1. 问题

你想把行数据变成以某种符号分隔的列表，例如以逗号分隔，而不是常见的竖排的列数据形式。你希望转换下面的结果集。

```
DEPTNO EMPS
------ ----------
    10 CLARK
    10 KING
    10 MILLER
    20 SMITH
    20 ADAMS
    20 FORD
    20 SCOTT
    20 JONES
    30 ALLEN
    30 BLAKE
    30 MARTIN
    30 JAMES
    30 TURNER
    30 WARD
```

变成这样：

```
 DEPTNO EMPS
------- ------------------------------------
     10 CLARK,KING,MILLER
     20 SMITH,JONES,SCOTT,ADAMS,FORD
     30 ALLEN,WARD,MARTIN,BLAKE,TURNER,JAMES
```

2. 解决方案

对于本问题而言，每一种数据库的解决方案都不同，关键在于如何充分利用数据库的内置函数。弄清楚数据库提供了哪些函数，我们才能充分利用数据库的功能，针对那些传统上 SQL 不擅长的问题探索出创造性的解决方案。

DB2

使用 `WITH` 子句递归地查询创建分隔列表。

```
 1   with x (deptno, cnt, list, empno, len)
 2     as (
 3 select deptno, count(*) over (partition by deptno),
 4        cast(ename as varchar(100)), empno, 1
 5   from emp
 6  union all
 7 select x.deptno, x.cnt, x.list ||','|| e.ename, e.empno, x.len+1
 8   from emp e, x
 9  where e.deptno = x.deptno
10    and e.empno > x. empno
11        )
12 select deptno,list
13   from x
14  where len = cnt
```

MySQL

使用内置函数 GROUP_CONCAT 创建分隔列表。

```
1 select deptno,
2        group_concat(ename order by empno separator, ',') as emps
3   from emp
4  group by deptno
```

Oracle

使用内置函数 SYS_CONNECT_BY_PATH 创建分隔列表。

```
 1  select deptno,
 2         ltrim(sys_connect_by_path(ename,','),',') emps
 3    from (
 4  select deptno,
 5         ename,
 6         row_number() over
 7                  (partition by deptno order by empno) rn,
 8         count(*) over
 9                  (partition by deptno) cnt
10    from emp
11         )
12   where level   = cnt
13   start with rn = 1
14 connect by prior deptno = deptno and prior rn = rn-1
```

PostgreSQL

PostgreSQL 没有提供用于创建分隔列表的标准内置函数，因而需要提前知道列表里有多少个元素。知道了最大的列表长度，就能确定在使用置换和字符串拼接等传统手段创建列表时需要附加多少个值。

```
 1 select deptno,
 2        rtrim(
 3            max(case when pos=1 then emps else '' end)||
 4            max(case when pos=2 then emps else '' end)||
 5            max(case when pos=3 then emps else '' end)||
 6            max(case when pos=4 then emps else '' end)||
 7            max(case when pos=5 then emps else '' end)||
 8            max(case when pos=6 then emps else '' end),','
 9        ) as emps
10   from (
11 select a.deptno,
12        a.ename||',' as emps,
13        d.cnt,
14        (select count(*) from emp b
15          where a.deptno=b.deptno and b.empno <= a.empno) as pos
16   from emp a,
17        (select deptno, count(ename) as cnt
18           from emp
19          group by deptno) d
20  where d.deptno=a.deptno
21        ) x
22  group by deptno
23  order by 1
```

SQL Server

使用 WITH 子句递归地查询创建分隔列表。

```
 1   with x (deptno, cnt, list, empno, len)
 2     as (
 3 select deptno, count(*) over (partition by deptno),
 4        cast(ename as varchar(100)),
 5        empno,
 6        1
 7   from emp
 8  union all
 9 select x.deptno, x.cnt,
10        cast(x.list + ',' + e.ename as varchar(100)),
11        e.empno, x.len+1
12   from emp e, x
13  where e.deptno = x.deptno
14    and e.empno > x. empno
15        )
16 select deptno,list
17   from x
18  where len = cnt
19  order by 1
```

3. 讨论

用 SQL 创建分隔列表之所以有用，是因为它是一个常见任务。然而，每种数据库的做法却各不相同。甚至，不同数据库的解决方案之间几乎没有相同之处。从使用递归、分层函数、经典的类型转换到数据聚合，各个数据库的做法迥异。

DB2 和 SQL Server

这两种数据库的解决方案仅在语法上略有不同（DB2 的字符串连接运算符是 ||，SQL Server 则是 +），具体做法完全相同。WITH 子句的第一个查询（UNION ALL 的前半部分）返回每位员工的下列信息：部门、员工编号、名字、ID 和常量 1（在这里，该常量没有任何作用）。递归处理发生在第二个查询（UNION ALL 的后半部分），并生成分隔列表。为了理解分隔列表的生成过程，我们来仔细观察该解决方案的一些代码片段。首先是 UNION ALL 的第二个查询的 SELECT 列表的第三项。

```
x.list ||','|| e.ename
```

然后是该查询的 WHERE 子句。

```
where e.deptno = x.deptno
  and e.empno > x.empno
```

本解决方案首先确保员工是同一个部门的。然后，对于 UNION ALL 的第一个查询返回的每一个员工，只要员工编号比自己大，就把名字附加在分隔列表的最后。这样就能确保不会把自己的名字附加到最后。表达式：

```
x.len+1
```

在每次一个员工被评估过之后为 LEN（从 1 开始）加上 1。如果累加值等于该部门的员工总数，我们就知道全部员工都被评估过了，分隔列表也就创建完成了。这是关键所在，它

不仅标志着分隔列表创建完成，也能及时终止递归处理。

```
where len = cnt
```

MySQL

GROUP_CONCAT 函数可以完成所有的工作。它负责把传递给它的 ENAME 列拼接起来。GROUP_CONCAT 函数是一个聚合函数，因而查询语句里需要用到 GROUP BY。

Oracle

理解 Oracle 解决方案的第一步是把它拆开来看。执行内嵌视图（第 4 ～ 10 行），生成的结果集包括每位员工的下列信息：部门、名字，按照 EMPNO 升序排列得出的每位员工在各自部门的排名，以及本部门的员工总数。例如：

```
select deptno,
       ename,
       row_number() over
                 (partition by deptno order by empno) rn,
       count(*) over (partition by deptno) cnt
  from emp

DEPTNO ENAME      RN CNT
------ ---------- -- ---
    10 CLARK       1   3
    10 KING        2   3
    10 MILLER      3   3
    20 SMITH       1   5
    20 JONES       2   5
    20 SCOTT       3   5
    20 ADAMS       4   5
    20 FORD        5   5
    30 ALLEN       1   6
    30 WARD        2   6
    30 MARTIN      3   6
    30 BLAKE       4   6
    30 TURNER      5   6
    30 JAMES       6   6
```

排名（上述查询里别名为 RN）是为了方便遍历整棵树。由于 ROW_NUMBER 生成从 1 开始的连续数字序列，不会有重复数字，也不会有空隙，因此如果想参照前一行（或者父节点），只需要（把当前的 RN 值）减 1 即可。例如，3 前面的数字是 3 减去 1，结果是 2。在这里，2 是 3 的父节点，可以通过第 12 行观察到这一点。除此之外，下面的两行代码：

```
start with rn = 1
connect by prior deptno = deptno
```

指明 RN 等于 1 的节点即为每个 DEPTNO 的根节点，并为（RN 等于 1 的记录出现时）每一个新出现的部门创建一个单独的列表。

这时，我们应该停下来，再看一下 ROW_NUMBER 函数的 ORDER BY 部分。请记住，员工名字会按照 EMPNO 排名，列表也按照 EMPNO 的顺序生成。每个部门的员工总数会被计算出来（别名为 CNT），并用于确保只有那些包含部门内全体员工的列表才会被返回。之所以要这样

做，是因为 SYS_CONNECT_BY_PATH 会循环生成列表，而我们也不希望得到不完整的列表。

对于层次查询，伪列 LEVEL 从 1 开始（对于不使用 CONNECT BY 的查询，LEVEL 是 0。但在 OracleDatabase 10*g* 数据库及后续版本里，LEVEL 必须和 CONNECT BY 同时出现），每当部门里的一个员工被评估（即层次遍历每深入一层）后，LEVEL 会加 1。因此，当 LEVEL 和 CNT 相等的时候，我们就知道循环走到了最后一个 EMPNO，这时就产生了一个完整的列表。



SYS_CONNECT_BY_PATH 函数会在列表的前面也加上一个事先选定的分隔符（本例中是逗号）。有时这不符合我们的预期。本实例的解决方案里，我们调用 LTRIM 函数删除了列表开头的逗号。

PostgreSQL

PostgreSQL 解决方案要求事先知道所有部门里员工总数的最大值。执行内嵌视图（第 11 ～ 18 行）生成的结果集包括（每位员工的）部门、后面附加了逗号的名字、所属部门的员工总数以及 EMPNO 比他小的员工总数。

```
deptno  |  emps    | cnt | pos
--------+---------+-----+-----
     20 | SMITH,  |   5 |   1
     30 | ALLEN,  |   6 |   1
     30 | WARD,   |   6 |   2
     20 | JONES,  |   5 |   2
     30 | MARTIN, |   6 |   3
     30 | BLAKE,  |   6 |   4
     10 | CLARK,  |   3 |   1
     20 | SCOTT,  |   5 |   3
     10 | KING,   |   3 |   2
     30 | TURNER, |   6 |   5
     20 | ADAMS,  |   5 |   4
     30 | JAMES,  |   6 |   6
     20 | FORD,   |   5 |   5
     10 | MILLER, |   3 |   3
```

产生 POS 列的标量子查询（第 14 ～ 15 行）用于按照 EMPNO 来排列每一个员工。例如，下面这行代码。

```
max(case when pos = 1 then emps else '' end)||
```

上述代码评估 POS 是否等于 1。如果 POS 等于 1，CASE 表达式将返回员工名字，反之则返回 Null。

我们必须先查询整个表的数据，搞清楚一个列表里最多可能出现几个值。对于 EMP 表的数据而言，一个部门里最多有 6 个人，因此一个列表里最多会出现 6 项。

下一步就是开始创建列表。我们在内嵌视图返回的行数据之上（以 CASE 表达式的形式）执行一些条件逻辑运算来做到这一点。

列表里可能出现多个值，我们就必须写多个 CASE 表达式。

如果 POS 等于 1，当前的名字会被加入列表。第二个 CASE 表达式评估 POS 是否等于 2；如果是，则第二个名字会被附加在第一个后面。如果没有第二个名字，则会有一个额外的逗号附

加在第一个名字后面（对于每一个不同的 POS 值，该处理都会重复一次，直至最后一个）。

这里的 MAX 函数调用不可省略，因为每个部门只需要生成一个列表（也可以调用 MIN 函数。本例中两者没有分别，因为对于每一个 CASE 条件运算，POS 只返回一个值）。无论何时调用了聚合函数，SELECT 列表里不涉及聚合运算的项目必须出现在 GROUP BY 子句里。这一点确保了每个 SELECT 列表里不涉及聚合运算的项目只会出现一行。

注意，我们需要 RTRIM 函数来删除末尾的逗号，逗号的数目总是等于列表里可能出现的值的最大个数（本例中为 6）。

6.11 分隔数据转换为多值IN列表

1. 问题

你有一些分隔数据，想传递给 WHERE 子句的 IN 列表。考虑下面的字符串。

```
7654,7698,7782,7788
```

你希望在 WHERE 子句里使用上述字符串，但是下面的 SQL 会由于 EMPNO 列是数值字段而执行失败。

```
select ename,sal,deptno
  from emp
 where empno in ( '7654,7698,7782,7788' )
```

上述 SQL 之所以失败是因为，EMPNO 列是数值类型，而 IN 列表里却只有一个字符串。你希望上述字符串能被当作逗号分隔的数值列表。

2. 解决方案

从表面上看，我们应该设法让 SQL 把分隔字符串当成一系列用逗号分隔好的值。然而，事实并非如此。如果一个逗号出现在引号里，SQL 无法知道它是一个多值列表。SQL 会把引号中的任何值当成单一的字符串数据。我们必须把该字符串打散，变成单个的 EMPNO。本解决方案的关键之处在于遍历字符串，但是并不需要深入到每一个字符。只需要遍历字符串中每一个有效的 EMPNO 即可。

DB2

遍历传递给 IN 列表的字符串，我们很容易将其转换为行数据。在这里，函数 ROW_NUMBER、LOCATE 和 SUBSTR 非常有用。

```
 1  select empno,ename,sal,deptno
 2    from emp
 3   where empno in (
 4  select cast(substr(c,2,locate(',',c,2)-2) as integer) empno
 5    from (
 6  select substr(csv.emps,cast(iter.pos as integer)) as c
 7    from (select ','||'7654,7698,7782,7788'||',' emps
 8            from t1) csv,
 9          (select id as pos
10            from t100 ) iter
11   where iter.pos <= length(csv.emps)
```

```
12          )  x
13    where length(c) > 1
14      and substr(c,1,1) = ','
15          )  y
```

MySQL

遍历传递给 IN 列表的字符串，我们很容易将其转换为行数据。

```
 1 select empno, ename, sal, deptno
 2   from emp
 3  where empno in
 4        (
 5 select substring_index(
 6        substring_index(list.vals,',',iter.pos),',',-1) empno
 6   from (select id pos from t10) as iter,
 7        (select '7654,7698,7782,7788' as vals
 8           from t1) list
 9  where iter.pos <=
10        (length(list.vals)-length(replace(list.vals,',','')))+1
11        )  x
```

Oracle

遍历传递给 IN 列表的字符串，我们很容易将其转换为行数据。在这里，ROWNUM、SUBSTR 和 INSTR 非常有用。

```
 1 select empno,ename,sal,deptno
 2   from emp
 3  where empno in (
 4        select to_number(
 5                   rtrim(
 6                  substr(emps,
 7                   instr(emps,',',1,iter.pos)+1,
 8                   instr(emps,',',1,iter.pos+1) -
 9                   instr(emps,',',1,iter.pos)),',')) emps
10          from (select ','||'7654,7698,7782,7788'||',' emps from t1) csv,
11                (select rownum pos from emp) iter
12          where iter.pos <= ((length(csv.emps)-
13                     length(replace(csv.emps,',')))/length(','))-1
14  )
```

PostgreSQL

遍历传递给 IN 列表的字符串，我们很容易将其转换为行数据。SPLIT_PART 函数能方便地把字符串解析成多个单独的数字。

```
 1 select ename,sal,deptno
 2   from emp
 3  where empno in (
 4 select cast(empno as integer) as empno
 5   from (
 6 select split_part(list.vals,',',iter.pos) as empno
 7   from (select id as pos from t10) iter,
 8        (select ','||'7654,7698,7782,7788'||',' as vals
 9           from t1) list
10  where iter.pos <=
```

```
11        length(list.vals)-length(replace(list.vals,',',''))
12        )  z
13  where length(empno) > 0
14        )  x
```

SQL Server

遍历传递给 IN 列表的字符串，我们很容易将其转换为行数据。在这里，函数 ROW_NUMBER、CHARINDEX 和 SUBSTRING 非常有用。

```
 1 select empno,ename,sal,deptno
 2   from emp
 3  where empno in (select substring(c,2,charindex(',',c,2)-2) as empno
 4   from (
 5 select substring(csv.emps,iter.pos,len(csv.emps)) as c
 6   from (select ','+'7654,7698,7782,7788'+',' as emps
 7           from t1) csv,
 8        (select id as pos
 9           from t100) iter
10  where iter.pos <= len(csv.emps)
11        )  x
12  where len(c) > 1
13    and substring(c,1,1) = ','
14        )  y
```

3. 讨论

这个解决方案的第一步就是遍历字符串，这也是最重要的一步。一旦你完成了这一步，那么剩下的就是解析字符串，用数据库提供的函数把字符串逐个转换为数值。

DB2 和 SQL Server

内嵌视图 X（第 6 ～ 11 行）遍历字符串。该解决方案的思路就是遍历字符串，因此每一行都比它前面的那行少一个字符。

```
,7654,7698,7782,7788,
7654,7698,7782,7788,
654,7698,7782,7788,
54,7698,7782,7788,
4,7698,7782,7788,
,7698,7782,7788,
7698,7782,7788,
698,7782,7788,
98,7782,7788,
8,7782,7788,
,7782,7788,
7782,7788,
782,7788,
82,7788,
2,7788,
,7788,
7788,
788,
88,
8,
,
```

注意，字符串前后都有逗号（分隔符），因此不需要特意检查字符串的起止位置。

下一步就是只保留我们想要放在 IN 列表中的值。这些值都以逗号开头，但是要排除掉最后一行，因为最后一行只有一个孤零零的逗号。调用函数 SUBSTR 或 SUBSTRING 筛选出以逗号开头的行，然后在那一行里找到下一个逗号，并留下两个逗号之间的所有字符。这一步完成后，接着要把找到的字符串转换为数字，这样就可以针对数值类型的 EMPNO 列（第 4 ～ 14 行）进行适当的评估。

```
 EMPNO
------
  7654
  7698
  7782
  7788
```

最后，把结果集放入一个子查询里，并返回想要得到的行。

MySQL

内嵌视图（第 5 ～ 9 行）遍历字符串。第 10 行的表达式决定了字符串里包含多少个值，这是通过找出字符串中有多少个逗号（分隔符）并加上 1 来实现的。函数 SUBSTRING_INDEX（第 6 行）返回字符串中第 *n* 次出现逗号（分隔符）之前（从左边开始）的所有字符。

```
+---------------------+
| empno               |
+---------------------+
| 7654                |
| 7654,7698           |
| 7654,7698,7782      |
| 7654,7698,7782,7788 |
+---------------------+
```

然后，上面得到的行会被再次传递给 SUBSTRING_INDEX 函数（第 5 行）。这一次的参数里，指定的第 *n* 次出现分隔符的参数值是 -1，这意味着从右边数第 *n* 次出现分隔符后，其右侧所有字符都会被保留下来。

```
+-------+
| empno |
+-------+
| 7654  |
| 7698  |
| 7782  |
| 7788  |
+-------+
```

最后，将上述结果放入一个子查询中。

Oracle

第一步是遍历字符串。

```
select emps,pos
  from (select ','||'7654,7698,7782,7788'||',' emps
          from t1) csv,
       (select rownum pos from emp) iter
```

```
 where iter.pos <=
 ((length(csv.emps)-length(replace(csv.emps,',')))/length(','))-1

EMPS                        POS
-------------------- ----------
,7654,7698,7782,7788,         1
,7654,7698,7782,7788,         2
,7654,7698,7782,7788,         3
,7654,7698,7782,7788,         4
```

上述查询返回的行数代表了列表中有多少个值。POS 列至关重要，有了它才能把字符串解析成单个的值。使用 SUBSTR 函数和 INSTR 函数解析字符串。POS 列被用来找出分隔符在每个字符串中第 *n* 次出现时的位置。由于字符串前后都有逗号，就不再需要做特别的检查来确定字符串的起止位置。被传递到函数 SUBSTR 和 INSTR（第 7 ～ 9 行）的值能找出分隔符第 *n* 次和第 *n*+1 次出现的位置。通过使用下一个逗号（在字符串中下一个逗号出现的位置）的返回值减去当前逗号（在字符串中当前逗号所在的位置）的返回值，我们就能从字符串中提取出每一个值。

```
select substr(emps,
       instr(emps,',',1,iter.pos)+1,
       instr(emps,',',1,iter.pos+1) -
       instr(emps,',',1,iter.pos)) emps
  from (select ','||'7654,7698,7782,7788'||',' emps
          from t1) csv,
       (select rownum pos from emp) iter
 where iter.pos <=
 ((length(csv.emps)-length(replace(csv.emps,',')))/length(','))-1

EMPS
-----------
7654,
7698,
7782,
7788,
```

最后，删除每个值后面的逗号，将其转换为数字并放入子查询中。

PostgreSQL

内嵌视图 Z（第 6 ～ 9 行）遍历字符串。返回的行数取决于字符串中含有多少个值。为了找出字符串中含有多少个值，用包含分隔符的字符串总长度减去去掉了分隔符的字符串长度（第 9 行）。SPLIT_PART 函数可以解析字符串，并找到分隔符第 *n* 次出现之前的那个值。

```
select list.vals,
       split_part(list.vals,',',iter.pos) as empno,
       iter.pos
  from (select id as pos from t10) iter,
       (select ','||'7654,7698,7782,7788'||',' as vals
          from t1) list
 where iter.pos <=
       length(list.vals)-length(replace(list.vals,',',''))

         vals          | empno | pos
```

```
----------------------+-------+-----
 ,7654,7698,7782,7788, |       |   1
 ,7654,7698,7782,7788, | 7654  |   2
 ,7654,7698,7782,7788, | 7698  |   3
 ,7654,7698,7782,7788, | 7782  |   4
 ,7654,7698,7782,7788, | 7788  |   5
```

最后，把这些值（EMPNO 列）转换成数字，并将其放入子查询中。

6.12　按字母表顺序排列字符

1. 问题

你想按照字母表顺序对字符串里的字符进行排序，考虑下面的结果集。

```
ENAME
----------
ADAMS
ALLEN
BLAKE
CLARK
FORD
JAMES
JONES
KING
MARTIN
MILLER
SCOTT
SMITH
TURNER
WARD
```

你希望得到如下所示的结果集。

```
OLD_NAME   NEW_NAME
---------- --------
ADAMS      AADMS
ALLEN      AELLN
BLAKE      ABEKL
CLARK      ACKLR
FORD       DFOR
JAMES      AEJMS
JONES      EJNOS
KING       GIKN
MARTIN     AIMNRT
MILLER     EILLMR
SCOTT      COSTT
SMITH      HIMST
TURNER     ENRRTU
WARD       ADRW
```

2. 解决方案

本问题是一个绝佳的例证，它表明为什么理解一种数据库并掌握其提供的各项功能是多么重要。如果我们正在使用的数据库没有提供合适的内置函数来帮我们解决问题，我们就需

要想一些别出心裁的办法。不妨比较下面的 MySQL 解决方案和其他数据库的解决方案。

DB2

为了对多行字符串进行排序，必须遍历每个字符串，然后对其中的字符进行排序。

```
 1  select ename,
 2         max(case when pos=1 then c else '' end)||
 3         max(case when pos=2 then c else '' end)||
 4         max(case when pos=3 then c else '' end)||
 5         max(case when pos=4 then c else '' end)||
 6         max(case when pos=5 then c else '' end)||
 7         max(case when pos=6 then c else '' end)
 8    from (
 9  select e.ename,
10         cast(substr(e.ename,iter.pos,1) as varchar(100)) c,
11         cast(row_number()over(partition by e.ename
12                              order by substr(e.ename,iter.pos,1))
13              as integer) pos
14    from emp e,
15         (select cast(row_number()over() as integer) pos
16            from emp) iter
17   where iter.pos <= length(e.ename)
18          ) x
19   group by ename
```

MySQL

这里的关键是 GROUP_CONCAT 函数，该函数不仅能连接员工名字字符串里的每个字符，还能对它们进行排序。

```
1  select ename, group_concat(c order by c separator '')
2    from (
3  select ename, substr(a.ename,iter.pos,1) c
4    from emp a,
5         ( select id pos from t10 ) iter
6   where iter.pos <= length(a.ename)
7         ) x
8   group by ename
```

Oracle

SYS_CONNECT_BY_PATH 函数能迭代地创建一个列表。

```
 1  select old_name, new_name
 2    from (
 3  select old_name, replace(sys_connect_by_path(c,' '),' ') new_name
 4    from (
 5  select e.ename old_name,
 6         row_number() over(partition by e.ename
 7                          order by substr(e.ename,iter.pos,1)) rn,
 8         substr(e.ename,iter.pos,1) c
 9    from emp e,
10         ( select rownum pos from emp ) iter
11   where iter.pos <= length(e.ename)
12   order by 1
13         ) x
```

```
14   start with rn = 1
15 connect by prior rn = rn-1 and prior old_name = old_name
16         )
17   where length(old_name) = length(new_name)
```

PostgreSQL

PostgreSQL 中没有能够方便地对字符串中的字符进行排序的内置函数，因此我们不仅要遍历每个字符串，还需要提前知道长度最大的员工名字。为了提高代码的可读性，本解决方案使用视图 V。

```
create or replace view V as
select x.*
  from (
select a.ename,
       substr(a.ename,iter.pos,1) as c
  from emp a,
       (select id as pos from t10) iter
 where iter.pos <= length(a.ename)
 order by 1,2
       ) x
```

下面的 SELECT 语句使用了上述视图。

```
 1 select ename,
 2        max(case when pos=1 then
 3                 case when cnt=1 then c
 4                      else rpad(c,cast(cnt as integer),c)
 5                 end
 6                 else ''
 7            end)||
 8        max(case when pos=2 then
 9                 case when cnt=1 then c
10                      else rpad(c,cast(cnt as integer),c)
11                 end
12                 else ''
13            end)||
14        max(case when pos=3 then
15                 case when cnt=1 then c
16                      else rpad(c,cast(cnt as integer),c)
17                 end
18                 else ''
19            end)||
20        max(case when pos=4 then
21                 case when cnt=1 then c
22                      else rpad(c,cast(cnt as integer),c)
23                 end
24                 else ''
25            end)||
26        max(case when pos=5 then
27                 case when cnt=1 then c
28                      else rpad(c,cast(cnt as integer),c)
29                 end
30                 else ''
31            end)||
32        max(case when pos=6 then
```

```
33                 case when cnt=1 then c
34                      else rpad(c,cast(cnt as integer),c)
35                 end
36                 else ''
37            end)
38   from (
39 select a.ename, a.c,
40        (select count(*)
41           from v b
42          where a.ename=b.ename and a.c=b.c ) as cnt,
43        (select count(*)+1
44           from v b
45          where a.ename=b.ename and b.c<a.c) as pos
46   from v a
47        ) x
48  group by ename
```

SQL Server

为了对多行字符串进行排序，必须遍历每个字符串，并对其中的字符进行排序。

```
 1  select ename,
 2            max(case when pos=1 then c else '' end)+
 3            max(case when pos=2 then c else '' end)+
 4            max(case when pos=3 then c else '' end)+
 5            max(case when pos=4 then c else '' end)+
 6            max(case when pos=5 then c else '' end)+
 7            max(case when pos=6 then c else '' end)
 8       from (
 9     select e.ename,
10           substring(e.ename,iter.pos,1) as c,
11           row_number() over (
12            partition by e.ename
13                order by substring(e.ename,iter.pos,1)) as pos
14       from emp e,
15            (select row_number()over(order by ename) as pos
16               from emp) iter
17      where iter.pos <= len(e.ename)
18            )  x
19      group by ename
```

3. 讨论

DB2 和 SQL Server

内嵌视图 X 把每个名字字符串里的每个字符都提取出来，并当作一行返回。函数 SUBSTR 或函数 SUBSTRING 提取出名字的每个字符，并且 ROW_NUMBER 函数按照字母表顺序排序每个字符。

```
ENAME     C    POS
-----     -    ---
ADAMS     A    1
ADAMS     A    2
ADAMS     D    3
ADAMS     M    4
ADAMS     S    5
...
```

为了把字符串中的每个字母都提取出来，并当作一行返回，我们必须遍历整个字符串。这个工作由内嵌视图 ITER 来完成。

现在，每个名字的字母都已经按照字母表顺序排列，最后一步就是将这些字母按顺序连接成新的字符串。CASE 语句（第 2 ~ 7 行）评估每个字母的位置。如果某个特定位置的字符被发现了，那么它就会被连接到下一个评估（后续的 CASE 语句）的结果里。因为调用了聚合函数 MAX，对应于每个 POS 值只有一个字符会被返回，因此对应于每个名字只会返回一行数据。CASE 语句一共有 6 个，这是因为 EMP 表里最长的名字只含有 6 个字符。

MySQL

内嵌视图 X（第 3 ~ 6 行）把每个名字的字符都提取出来，并当作一行返回。SUBSTR 函数可以提取名字字符串里的每个字符。

```
ENAME    C
-----    -
ADAMS    A
ADAMS    A
ADAMS    D
ADAMS    M
ADAMS    S
...
```

内嵌视图 ITER 用于遍历字符串。其余的工作都交由 GROUP_CONCAT 函数完成。通过指定排序方式，GROUP_CONCAT 函数不仅能串接每个字母，还能按照字母表顺序对它们进行排序。

Oracle

最重要的工作是由视图 X（第 5 ~ 11 行）完成的，它提取出每个名字的字符，并按照字母表顺序排列好。通过遍历字符串并对字符执行排序实现了这一点。查询语句的剩余部分只是将排好序的名字字符粘结到一起而已。

只执行内嵌视图 X 的话，就能看到把名字拆解之后得到的各个字符了。

```
OLD_NAME           RN C
---------- ---------- -
ADAMS               1 A
ADAMS               2 A
ADAMS               3 D
ADAMS               4 M
ADAMS               5 S
...
```

然后，提取出排好序的字符并重建每个名字。可以使用 SYS_CONNECT_BY_PATH 函数来完成这一步，它把所有的字符按顺序串接起来。

```
OLD_NAME   NEW_NAME
---------- ----------
ADAMS      A
ADAMS      AA
ADAMS      AAD
ADAMS      AADM
ADAMS      AADMS
...
```

最后，只保留那些和原名字具有相同长度的字符串。

PostgreSQL

为了提高可读性，本解决方案使用视图 V 来遍历字符串。视图里的 SUBSTR 函数会提取每个名字的全部字符，得到如下结果集。

```
ENAME    C
-----    -
ADAMS    A
ADAMS    A
ADAMS    D
ADAMS    M
ADAMS    S
...
```

该视图也按照 ENAME 和从每个名字提取出来的字母做了排序。内嵌视图 X（第 15 ～ 18 行）从视图 V 里检索出名字、字符、字符在名字里出现的次数及其位置（按字母表顺序排列）。

```
 ename | c | cnt | pos
-------+---+-----+-----
 ADAMS | A |   2 |   1
 ADAMS | A |   2 |   1
 ADAMS | D |   1 |   3
 ADAMS | M |   1 |   4
 ADAMS | S |   1 |   5
```

对于该解决方案而言，由内嵌视图 X 返回的 CNT 列和 POS 列非常重要。POS 列用来对每个字符进行排序，而 CNT 列用于确定每个字符在名字中出现的次数。最后一步是，评估每个字符的位置并重建名字。注意，每个 CASE 语句实际上都包含两个 CASE 子句。这是为了确认一个字符在名字中是否出现了不止一次。如果出现多次的话，那么返回的就不是该字符，而是由 CNT 个该字符串接而成的字符串。聚合函数 MAX 能确保每个名字只返回一行数据。

6.13 识别字符串里的数字字符

1. 问题

你有一列包含字符的数据。不过，这些数据里不仅有数字，还有其他字符。考虑下面的视图 V。

```
create view V as
select replace(mixed,' ','') as mixed
  from (
select substr(ename,1,2)||
       cast(deptno as char(4))||
       substr(ename,3,2) as mixed
  from emp
 where deptno = 10
 union all
select cast(empno as char(4)) as mixed
  from emp
 where deptno = 20
```

```
 union all
select ename as mixed
  from emp
 where deptno = 30
       ) x

select * from v

MIXED
--------------
CL10AR
KI10NG
MI10LL
7369
7566
7788
7876
7902
ALLEN
WARD
MARTIN
BLAKE
TURNER
JAMES
```

你希望筛选出只包含数字或至少有一个数字的行。如果既有数字又有其他字符，你希望删除非数字字符，只返回数字。对于以上示例数据而言，你希望得到下面的结果集。

```
   MIXED
--------
      10
      10
      10
    7369
    7566
    7788
    7876
    7902
```

2. 解决方案

函数 REPLACE 和 TRANSLATE 对于操作字符串和单个字符非常有用。关键在于把全部数字替换为某个字符，这样就能通过读取该字符的方式很方便地隔离和识别数字字符。

DB2

使用函数 TRANSLATE、REPLACE 和 POSSTR 分离出每一行的数字字符。还需要在视图 V 里调用 CAST。否则的话，会因为类型转换错误导致视图创建失败。你需要使用 REPLACE 函数删除无关的空白字符，以便转换为固定长度的 CHAR 类型。

```
1 select mixed old,
2        cast(
3          case
4          when
5             replace(
```

```
 6          translate(mixed,'9999999999','0123456789'),'9','') = ''
 7         then
 8            mixed
 9         else replace(
10            translate(mixed,
11               repeat('#',length(mixed)),
12             replace(
13              translate(mixed,'9999999999','0123456789'),'9','')),
14                      '#','')
15          end as integer ) mixed
16   from V
17  where posstr(translate(mixed,'9999999999','0123456789'),'9') > 0
```

MySQL

MySQL 的语法略有不同，下面是视图 V 的定义。

```
create view V as
select concat(
         substr(ename,1,2),
         replace(cast(deptno as char(4)),' ',''),
         substr(ename,3,2)
       ) as mixed
  from emp
 where deptno = 10
 union all
select replace(cast(empno as char(4)), ' ', '')
  from emp where deptno = 20
 union all
select ename from emp where deptno = 30
```

由于 MySQL 不支持 TRANSLATE 函数，我们必须遍历每一行字符串，并评估每一个字符。

```
 1 select cast(group_concat(c order by pos separator '') as unsigned)
 2        as MIXED1
 3   from (
 4 select v.mixed, iter.pos, substr(v.mixed,iter.pos,1) as c
 5   from V,
 6        ( select id pos from t10 ) iter
 7  where iter.pos <= length(v.mixed)
 8    and ascii(substr(v.mixed,iter.pos,1)) between 48 and 57
 9        ) y
10  group by mixed
11  order by 1
```

Oracle

使用函数 TRANSLATE、REPLACE 和 INSTR 分离出每一行的数字字符。在视图 V 中调用 CAST 不是必须的。使用 REPLACE 函数删除无关的空白字符，以便转换为固定长度的 CHAR 类型。如果希望在视图的定义里使用显式类型转换，不妨转换为 VARCHAR2 类型。

```
1 select to_number (
2        case
3        when
4           replace(translate(mixed,'0123456789','9999999999'),'9')
5          is not null
```

```
 6        then
 7             replace(
 8           translate(mixed,
 9             replace(
10          translate(mixed,'0123456789','9999999999'),'9'),
11                    rpad('#',length(mixed),'#')),'#')
12        else
13              mixed
14        end
15        ) mixed
16  from V
17 where instr(translate(mixed,'0123456789','9999999999'),'9') > 0
```

PostgreSQL

使用函数 TRANSLATE、REPLACE 和 STRPOS 分离出每一行的数字字符。在视图 V 中调用 CAST 不是必须的。使用 REPLACE 函数删除无关的空白字符，以便转换为固定长度的 CHAR 类型。如果希望在视图的定义里使用显式类型转换，建议转换为 VARCHAR2 类型。

```
 1 select cast(
 2        case
 3        when
 4         replace(translate(mixed,'0123456789','9999999999'),'9','')
 5         is not null
 6        then
 7           replace(
 8          translate(mixed,
 9            replace(
10          translate(mixed,'0123456789','9999999999'),'9',''),
11                    rpad('#',length(mixed),'#')),'#','')
12        else
13          mixed
14        end as integer ) as mixed
15    from V
16  where strpos(translate(mixed,'0123456789','9999999999'),'9') > 0
```

SQL Server

使用内置函数 ISNUMERIC 和通配符搜索能很容易识别出含有数字的字符串，但是 SQL Server 不支持 TRANSLATE 函数，我们无法高效地从字符串里提取数字字符。

3. 讨论

TRANSLATE 函数在这里非常有用，有了它，我们很容易把数字字符从其他字符分离识别出来。关键在于把所有数字先替换为某个字符。这样的话，我们只需要搜索一个既定字符，而不必去匹配不同的数字。

DB2、Oracle 和 PostgreSQL

这几种数据库的语法稍有不同，但方法相同。这里我选择讨论 PostgreSQL 的解决方案。

真正的工作是由函数 TRANSLATE 和 REPLACE 完成的。为了得到最终的结果集，需要多次使用这两个函数，下面的查询包含了每一次使用函数的情况。

```
select mixed as orig,
translate(mixed,'0123456789','9999999999') as mixed1,
```

```
replace(translate(mixed,'0123456789','9999999999'),'9','') as mixed2,
 translate(mixed,
 replace(
 translate(mixed,'0123456789','9999999999'),'9',''),
          rpad('#',length(mixed),'#')) as mixed3,
 replace(
 translate(mixed,
 replace(
translate(mixed,'0123456789','9999999999'),'9',''),
         rpad('#',length(mixed),'#')),'#','') as mixed4
  from V
 where strpos(translate(mixed,'0123456789','9999999999'),'9') > 0
```

```
  ORIG  | MIXED1 | MIXED2 | MIXED3 | MIXED4 | MIXED5
--------+--------+--------+--------+--------+--------
 CL10AR | CL99AR | CLAR   | ##10## | 10     |     10
 KI10NG | KI99NG | KING   | ##10## | 10     |     10
 MI10LL | MI99LL | MILL   | ##10## | 10     |     10
 7369   | 9999   |        | 7369   | 7369   |   7369
 7566   | 9999   |        | 7566   | 7566   |   7566
 7788   | 9999   |        | 7788   | 7788   |   7788
 7876   | 9999   |        | 7876   | 7876   |   7876
 7902   | 9999   |        | 7902   | 7902   |   7902
```

首先要注意到，不包含任何数字的行会被删掉。仔细阅读上述结果集的每一列之后，我们就会理解其工作原理。被筛选出来的行包括 ORIG 列的值和构成最终结果集的行。首先，调用 TRANSLATE 函数把所有数字都转换为 9（事实上，我们可以选择任何数字，这里的 9 是任意选的），这一结果就是 MIXED1 列的值。现在所有数字都变成了 9，它们能被当作一种单一字符来处理。然后，通过调用 REPLACE 函数删除所有的数字。由于数字都已经被替换成了 9，REPLACE 函数只需简单地搜索 9，就可以删除它们。这一结果用 MIXED2 列来表示。下一步，为了计算 MIXED3，需要使用 MIXED2 的返回值。把 MIXED2 列和 ORIG 列进行比较。如果 ORIG 匹配到了 MIXED2 的任意字符，那么就调用 TRANSLATE 函数将其替换为 #。MIXED3 列的结果显示出这些字母（而不是数字）已经被挑出来，并转换为某个字符（#）。现在所有非数字字符都变成了 #，因而它们也能被当作一种单一字符来处理。下一步，为了得到 MIXED4，调用 REPLACE 函数找到并删除每一行里的 # 字，剩余的部分就只有数字。最后，把数值字符转换为数字形式。现在已经完成了所有步骤，我们也就明白了 WHERE 子句的工作原理。MIXED1 的结果被传递给 STRPOS 函数，如果发现了一个 9（定位字符串中第一次出现 9 的位置），那么函数返回值一定大于 0。也就是说，若返回值大于 0，那么意味着该行至少存在一个数字，所以这一行应该被保留下来。

MySQL

首先遍历每个字符串，评估每个字符并判断其是否为数字。

```
select v.mixed, iter.pos, substr(v.mixed,iter.pos,1) as c
  from V,
     ( select id pos from t10 ) iter
 where iter.pos <= length(v.mixed)
 order by 1,2
```

```
+--------+------+------+
| mixed  | pos  | c    |
+--------+------+------+
| 7369   |    1 | 7    |
| 7369   |    2 | 3    |
| 7369   |    3 | 6    |
| 7369   |    4 | 9    |
...
| ALLEN  |    1 | A    |
| ALLEN  |    2 | L    |
| ALLEN  |    3 | L    |
| ALLEN  |    4 | E    |
| ALLEN  |    5 | N    |
...

| CL10AR |    1 | C    |
| CL10AR |    2 | L    |
| CL10AR |    3 | 1    |
| CL10AR |    4 | 0    |
| CL10AR |    5 | A    |
| CL10AR |    6 | R    |
+--------+------+------+
```

现在可以单独评估字符串中的每个字符，接下来需要筛选出在 C 列中有一个数字的行。

```
select v.mixed, iter.pos, substr(v.mixed,iter.pos,1) as c
  from V,
      ( select id pos from t10 ) iter
 where iter.pos <= length(v.mixed)
   and ascii(substr(v.mixed,iter.pos,1)) between 48 and 57
 order by 1,2

+--------+------+------+
| mixed  | pos  | c    |
+--------+------+------+
| 7369   |    1 | 7    |
| 7369   |    2 | 3    |
| 7369   |    3 | 6    |
| 7369   |    4 | 9    |
...
| CL10AR |    3 | 1    |
| CL10AR |    4 | 0    |
...
+--------+------+------+
```

到这一步，C 列只剩下数字。下面调用 GROUP_CONCAT 函数串接这些数字，并形成 MIXED 列的数字。需要将最后的结果转换为数字类型。

```
select cast(group_concat(c order by pos separator '') as unsigned)
         as MIXED1
  from (
select v.mixed, iter.pos, substr(v.mixed,iter.pos,1) as c
  from V,
      ( select id pos from t10 ) iter
 where iter.pos <= length(v.mixed)
```

```
and ascii(substr(x.mixed,iter.pos,1)) between 48 and 57
     ) y
group by mixed
order by 1

+--------+
| MIXED1 |
+--------+
|    10  |
|    10  |
|    10  |
|  7369  |
|  7566  |
|  7788  |
|  7876  |
|  7902  |
+--------+
```

最后要注意，每个字符串里的所有数字字符都会被串接成一个新数字。例如，输入值 99Gennick87 会导致数字 9987 被返回。这一点需要特别注意，尤其是处理序列化数据的时候。

6.14 提取第*n*个分隔子字符串

1. 问题

你想从一个字符串里提取出特定的分隔子字符串。考虑下面的视图 V，它生成了本问题的源数据。

```
create view V as
select 'mo,larry,curly' as name
  from t1
 union all
select 'tina,gina,jaunita,regina,leena' as name
  from t1
```

上述视图输出如下所示的数据。

```
select * from v

NAME
-------------------
mo,larry,curly
tina,gina,jaunita,regina,leena
```

你希望提取每一行的第二个名字，并得到下面这样的结果集。

```
  SUB
-----
larry
 gina
```

2. 解决方案

解决这一问题的关键是，把每一个名字转换为单独的一行，并保持每一个名字在列表里的

顺序不变。具体方法取决于你所使用的数据库。

DB2

遍历视图 V 返回的 NAME，并使用 ROW_NUMBER 函数筛选出每一个字符串里的第二个名字。

```
 1 select substr(c,2,locate(',',c,2)-2)
 2   from (
 3 select pos, name, substr(name, pos) c,
 4        row_number() over(partition by name
 5                         order by length(substr(name,pos)) desc) rn
 6   from (
 7 select ',' ||csv.name|| ',' as name,
 8        cast(iter.pos as integer) as pos
 9   from V csv,
10        (select row_number() over() pos from t100 ) iter
11  where iter.pos <= length(csv.name)+2
12        )  x
13  where length(substr(name,pos)) > 1
14    and substr(substr(name,pos),1,1) = ','
15        ) y
16  where rn = 2
```

MySQL

遍历视图 V 返回的 NAME，并使用逗号的位置来筛选出每一个字符串里的第二个名字。

```
 1 select name
 2   from (
 3 select iter.pos,
 4        substring_index(
 5        substring_index(src.name,',',iter.pos),',',-1) name
 6   from V src,
 7        (select id pos from t10) iter,
 8  where iter.pos <=
 9        length(src.name)-length(replace(src.name,',',''))
10        ) x
11 where pos = 2
```

Oracle

遍历视图 V 返回的 NAME，并使用 SUBSTR 函数和 INSTR 函数提取每个列表里的第二个名字。

```
 1 select sub
 2   from (
 3 select iter.pos,
 4        src.name,
 5        substr( src.name,
 6         instr( src.name,',',1,iter.pos )+1,
 7         instr( src.name,',',1,iter.pos+1 ) -
 8         instr( src.name,',',1,iter.pos )-1) sub
 9   from (select ','||name||',' as name from V) src,
10        (select rownum pos from emp) iter
11  where iter.pos < length(src.name)-length(replace(src.name,','))
12        )
13  where pos = 2
```

PostgreSQL

使用 SPLIT_PART 函数把每一个单独的名字作为一行返回。

```
 1 select name
 2   from (
 3 select iter.pos, split_part(src.name,',',iter.pos) as name
 4   from (select id as pos from t10) iter,
 5        (select cast(name as text) as name from v) src
 7  where iter.pos <=
 8         length(src.name)-length(replace(src.name,',',''))+1
 9        ) x
10  where pos = 2
```

SQL Server

遍历视图 V 返回的 NAME，并使用 ROW_NUMBER 函数筛选出每一个字符串里的第二个名字。

```
 1 select substring(c,2,charindex(',',c,2)-2)
 2   from (
 3 select pos, name, substring(name, pos, len(name)) as c,
 4        row_number() over(
 5         partition by name
 6         order by len(substring(name,pos,len(name))) desc) rn
 7   from (
 8 select ',' + csv.name + ',' as name,
 9         iter.pos
10   from V csv,
11        (select id as pos from t100 ) iter
12  where iter.pos <= len(csv.name)+2
13        ) x
14  where len(substring(name,pos,len(name))) > 1
15    and substring(substring(name,pos,len(name)),1,1) = ','
16        ) y
17  where rn = 2
```

3. 讨论

DB2 和 SQL Server

这两种数据库的解决方案的语法稍有不同，但方法相同。后面的讨论里我们以 DB2 数据库的解决方案为主。使用内嵌视图 X 遍历字符串，结果如下所示。

```
select  ','||csv.name|| ',' as name,
         iter.pos
  from v csv,
        (select row_number() over() pos from t100 ) iter
 where iter.pos <= length(csv.name)+2

EMPS                                 POS
------------------------------     ----
,tina,gina,jaunita,regina,leena,        1
,tina,gina,jaunita,regina,leena,        2
,tina,gina,jaunita,regina,leena,        3
...
```

然后遍历字符串中的每一个字符。

```
select pos, name, substr(name, pos) c,
        row_number() over(partition by name
                          order by length(substr(name, pos)) desc) rn
  from (
select ','||csv.name||',' as name,
       cast(iter.pos as integer) as pos
  from v csv,
      (select row_number() over() pos from t100 ) iter
 where iter.pos <= length(csv.name)+2
       )  x
 where length(substr(name,pos)) > 1

POS  NAME              C                 RN
---  --------------  ----------------  --
  1  ,mo,larry,curly,  ,mo,larry,curly,   1
  2  ,mo,larry,curly,  mo,larry,curly,    2
  3  ,mo,larry,curly,  o,larry,curly,     3
  4  ,mo,larry,curly,  ,larry,curly,      4
  ...
```

现在，我们得到了含有字符串不同部分的数据，并且很容易筛选出要保留的行。我们感兴趣的行都以逗号开头，其余的行都将被舍弃。

```
select pos, name, substr(name,pos) c,
       row_number() over(partition by name
                         order by length(substr(name, pos)) desc) rn
  from (
select ','||csv.name||',' as name,
       cast(iter.pos as integer) as pos
  from v csv,
        (select row_number() over() pos from t100 ) iter
 where iter.pos <= length(csv.name)+2
        )  x
 where length(substr(name,pos)) > 1
    and substr(substr(name,pos),1,1) = ','

POS   NAME                                C                                 RN
 ---   -------------                      ---------------                   --
  1   ,mo,larry,curly,                    ,mo,larry,curly,                   1
  4   ,mo,larry,curly,                    ,larry,curly,                      2
 10   ,mo,larry,curly,                    ,curly,                            3
  1   ,tina,gina,jaunita,regina,leena,   ,tina,gina,jaunita,regina,leena,   1
  6   ,tina,gina,jaunita,regina,leena,   ,gina,jaunita,regina,leena,        2
 11   ,tina,gina,jaunita,regina,leena,   ,jaunita,regina,leena,             3
 19   ,tina,gina,jaunita,regina,leena,   ,regina,leena,                     4
 26   ,tina,gina,jaunita,regina,leena,   ,leena,                            5
```

这是确定如何得到第 *n* 个子字符串的重要一步。注意，由于如下所示的 WHERE 条件，许多行已经被删除。

```
substr(substr(name,pos),1,1) = ','
```

注意，,larry,curly 这个字符串的原排名为 4，现在的排名却变成了 2。由于 WHERE 子句会在 SELECT 之前执行，因此以逗号开头的行会先被筛选出来，之后才调用 ROW_NUMBER 函数

决定每一行的编号。此时可以很清楚地看到，要得到第 *n* 个子字符串，只需要在 WHERE 子句里指定 RN 等于 n 即可。最后，只保留我们感兴趣的行（本例中是 RN 等于 2 的行），并调用 SUBSTR 函数提取那一行的名字。最后留下来的是每行中的第一个名字：,larry,curly, 里的 larry 和 ,gina,jaunita,regina,leena, 里的 gina。

MySQL

使用内嵌视图 X 遍历每个字符串，我们可以通过计算字符串中的分隔符的个数来确定该字符串中有多少个值。

```
select iter.pos, src.name
  from (select id pos from t10) iter,
       V src
 where iter.pos <=
       length(src.name)-length(replace(src.name,',',''))
```

```
+------+------------------------------+
| pos  | name                         |
+------+------------------------------+
|    1 | mo,larry,curly               |
|    2 | mo,larry,curly               |
|    1 | tina,gina,jaunita,regina,leena |
|    2 | tina,gina,jaunita,regina,leena |
|    3 | tina,gina,jaunita,regina,leena |
|    4 | tina,gina,jaunita,regina,leena |
+------+------------------------------+
```

上述查询结果中，每个字符串对应的数据行相较于字符串里实际的值的个数少了一行，因为这就是我们需要的。SUBSTRING_INDEX 函数可以解析我们需要的这些值。

```
select iter.pos,src.name name1,
        substring_index(src.name,',',iter.pos) name2,
        substring_index(
         substring_index(src.name,',',iter.pos),',',-1) name3
  from (select id pos from t10) iter,
       V src
 where iter.pos <=
       length(src.name)-length(replace(src.name,',',''))
```

```
+------+--------------------------------+--------------------------+---------+
| pos  | name1                          | name2                    | name3   |
+------+--------------------------------+--------------------------+---------+
|    1 | mo,larry,curly                 | mo                       | mo      |
|    2 | mo,larry,curly                 | mo,larry                 | larry   |
|    1 | tina,gina,jaunita,regina,leena | tina                     | tina    |
|    2 | tina,gina,jaunita,regina,leena | tina,gina                | gina    |
|    3 | tina,gina,jaunita,regina,leena | tina,gina,jaunita        | jaunita |
|    4 | tina,gina,jaunita,regina,leena | tina,gina,jaunita,regina | regina  |
+------+--------------------------------+--------------------------+---------+
```

我已经展示了 3 个和名字相关的字段，我们可以据此了解嵌套的 SUBSTRING_INDEX 函数是如何工作的。内层的函数能够找到逗号第 *n* 次出现的位置，并提取该位置左侧的全部字符。外层的函数可以找到（从字符串的末尾开始计数）逗号第一次出现的位置，并提取其

右侧的全部字符。最后，将 POS 等于 *n* 的 NAME3 值保留下来，本例中 *n* 为 2。

Oracle

使用内嵌视图遍历每一个字符串。字符串在内嵌视图结果集里出现的次数取决于字符串里有多少个值。本解决方案通过计算字符串里分隔符的出现次数得到每个字符串含有多少个值。因为字符串前后都有逗号，字符串里值的个数等于逗号出现的次数减 1。然后，这些字符串与另一个表合并，并执行连接查询，该表的记录条数至少要等于全体字符串中值的个数的最大值。函数 SUBSTR 和 INSTR 利用 POS 值解析每个字符串。

```
select iter.pos, src.name,
       substr( src.name,
        instr( src.name,',',1,iter.pos )+1,
        instr( src.name,',',1,iter.pos+1 ) -
        instr( src.name,',',1,iter.pos )-1) sub
  from (select ','||name||',' as name from v) src,
       (select rownum pos from emp) iter
 where iter.pos < length(src.name)-length(replace(src.name,','))

POS NAME                              SUB
--- --------------------------------- -------------
  1 ,mo,larry,curly,                  mo
  1 , tina,gina,jaunita,regina,leena,  tina
  2 ,mo,larry,curly,                  larry
  2 , tina,gina,jaunita,regina,leena, gina
  3 ,mo,larry,curly,                  curly
  3 , tina,gina,jaunita,regina,leena, jaunita
  4 , tina,gina,jaunita,regina,leena, regina
  5 , tina,gina,jaunita,regina,leena, leena
```

第一次调用 SUBSTR 函数中的 INSTR 函数可以确定要提取的子字符串的开始位置。第二次调用 SUBSTR 函数中的 INSTR 函数能够找到第 *n* 个逗号的位置（与开始位置相同）和第 *n*+1 个逗号的位置。上述两个值相减得到了要提取的字符串的长度。因为每个值被解析后都作为单独的行返回，只需要简单地指定 WHERE POS = n，就能筛选出第 *n* 个子字符串（本例中，WHERE POS = 2，因此要提取的是列表中第 2 个子字符串）。

PostgreSQL

使用内嵌视图 X 遍历每个字符串。返回的行数取决于每个字符串中包含多少个值。为了得到字符串里值的个数，我们需要计算出字符串中分隔符出现的次数，然后再加上 1。函数 SPLIT_PART 使用 POS 的值找到分隔符第 *n* 次出现的位置，并解析字符串提取出的名字。

```
select iter.pos, src.name as name1,
        split_part(src.name,',',iter.pos) as name2
   from (select id as pos from t10) iter,
        (select cast(name as text) as name from v) src
  where iter.pos <=
        length(src.name)-length(replace(src.name,',',''))+1

 pos |             name1              |  name2
-----+--------------------------------+---------
   1 | mo,larry,curly                 | mo
   2 | mo,larry,curly                 | larry
```

```
  3 | mo,larry,curly                 | curly
  1 | tina,gina,jaunita,regina,leena | tina
  2 | tina,gina,jaunita,regina,leena | gina
  3 | tina,gina,jaunita,regina,leena | jaunita
  4 | tina,gina,jaunita,regina,leena | regina
  5 | tina,gina,jaunita,regina,leena | leena
```

我展示了两遍 NAME 列，是为了说明 SPLIT_PART 函数是如何借助 POS 解析每个字符串的。一旦所有字符串都被解析过了，最后一步就是筛选 POS 等于我们感兴趣的第 *n* 个子字符串所在的行，本例中为 2。

6.15 解析IP地址

1. 问题

你想把一个 IP 地址的各个字段分解为四列，考虑下面的 IP 地址。

```
111.22.3.4
```

你希望查询语句能返回如下所示的结果。

```
A     B     C     D
----- ----- ----- ---
111   22    3     4
```

2. 解决方案

具体的解决方案取决于数据库提供的内置函数。不管是哪种数据库，关键之处都在于如何快速定位英文句号以及英文句号前后的数字。

DB2

使用 WITH 子句递归地查询针对 IP 地址的循环操作，同时使用 SUBSTR 函数可以很容易解析 IP 地址。在 IP 地址开头加上一个英文句号，这样每一组数字的开头位置都有英文句号，因而我们能以相同的方式处理所有的四组数字。

```
 1 with x (pos,ip) as (
 2   values (1,'.92.111.0.222')
 3   union all
 4  select pos+1,ip from x where pos+1 <= 20
 5 )
 6  select max(case when rn=1 then e end) a,
 7         max(case when rn=2 then e end) b,
 8         max(case when rn=3 then e end) c,
 9         max(case when rn=4 then e end) d
10    from (
11  select pos,c,d,
12         case when posstr(d,'.') > 0 then substr(d,1,posstr(d,'.')-1)
13              else d
14         end as e,
15         row_number() over(order by pos desc) rn
16    from (
17  select pos, ip,right(ip,pos) as c, substr(right(ip,pos),2) as d
18    from x
```

```
19   where pos <= length(ip)
20     and substr(right(ip,pos),1,1) = '.'
21        ) x
22        ) y
```

MySQL

使用函数 SUBSTR_INDEX 很容易解析 IP 地址。

```
1 select substring_index(substring_index(y.ip,'.',1),'.',-1) a,
2        substring_index(substring_index(y.ip,'.',2),'.',-1) b,
3        substring_index(substring_index(y.ip,'.',3),'.',-1) c,
4        substring_index(substring_index(y.ip,'.',4),'.',-1) d
5   from (select '92.111.0.2' as ip from t1) y
```

Oracle

使用内置函数 SUBSTR 和 INSTR 解析和遍历 IP 地址。

```
1 select ip,
2        substr(ip, 1, instr(ip,'.')-1 ) a,
3        substr(ip, instr(ip,'.')+1,
4                   instr(ip,'.',1,2)-instr(ip,'.')-1 ) b,
5        substr(ip, instr(ip,'.',1,2)+1,
6                   instr(ip,'.',1,3)-instr(ip,'.',1,2)-1 ) c,
7        substr(ip, instr(ip,'.',1,3)+1 ) d
8   from (select '92.111.0.2' as ip from t1)
```

PostgreSQL

使用内置函数 SPLIT_PART 解析 IP 地址。

```
1 select split_part(y.ip,'.',1) as a,
2        split_part(y.ip,'.',2) as b,
3        split_part(y.ip,'.',3) as c,
4        split_part(y.ip,'.',4) as d
5   from (select cast('92.111.0.2' as text) as ip from t1) as y
```

SQL Server

使用 WITH 子句递归地查询针对 IP 地址的循环操作，同时使用 SUBSTR 函数可以很容易解析 IP 地址。在 IP 地址开头加上英文句号，这样每一组数字的开头位置都有英文句号，因而我们能以相同的方式处理全部 4 组数字。

```
 1   with x (pos,ip) as (
 2     select 1 as pos,'.92.111.0.222' as ip from t1
 3     union all
 4    select pos+1,ip from x where pos+1 <= 20
 5   )
 6   select max(case when rn=1 then e end) a,
 7          max(case when rn=2 then e end) b,
 8          max(case when rn=3 then e end) c,
 9          max(case when rn=4 then e end) d
10     from (
11   select pos,c,d,
12          case when charindex('.',d) > 0
13               then substring(d,1,charindex('.',d)-1)
14               else d
```

```
15         end as e,
16         row_number() over(order by pos desc) rn
17    from (
18  select pos, ip,right(ip,pos) as c,
19         substring(right(ip,pos),2,len(ip)) as d
20    from x
21  where pos <= len(ip)
22    and substring(right(ip,pos),1,1) = '.'
23       ) x
24       ) y
```

3. 讨论

有了数据库提供的内置函数，我们很容易遍历字符串的各个部分。关键之处在于如何定位 IP 地址里英文句号。然后，我们就能解析英文句号之间的数字。

第 7 章

数值处理

本章主要介绍涉及数字的常见操作，包括数值计算。尽管对于复杂的数值计算而言，SQL 并非首选工具，但它足以胜任日常数值处理工作。



本章中的一些实例使用了聚合函数和 GROUP BY 子句。如果你不熟悉 SQL 的分组操作，请先阅读 A.1 节。

7.1 计算平均值

1. 问题

你想计算某一列的平均值，可能针对表中的全部记录，也可能只针对部分记录。例如，你希望知道全部员工的平均工资，同时也想知道每个部门的平均工资。

2. 解决方案

为了计算所有员工的平均工资，只需要针对 SAL 列调用 AVG 函数即可。由于没有使用 WHERE 子句，因此 AVG 函数会计算所有非 Null 值的平均值。

```
1 select avg(sal) as avg_sal
2   from emp

   AVG_SAL
----------
2073.21429
```

为了计算每个部门的平均工资，需要使用 GROUP BY 子句来对部门进行分组。

```
1 select deptno, avg(sal) as avg_sal
2   from emp
3  group by deptno
```

```
    DEPTNO    AVG_SAL
---------- ----------
        10 2916.66667
        20       2175
        30 1566.66667
```

3. 讨论

针对整个表计算平均值时，只要对感兴趣的列执行 AVG 函数即可，而且不需要使用 GROUP BY 子句。请记住，AVG 函数会忽略 Null。下面的例子演示了 Null 被忽略之后的结果。

```
create table t2(sal integer)
insert into t2 values (10)
insert into t2 values (20)
insert into t2 values (null)

select avg(sal)    select distinct 30/2
  from t2            from t2

  AVG(SAL)               30/2
----------         ----------
        15                 15

select avg(coalesce(sal,0))    select distinct 30/3
  from t2                        from t2

AVG(COALESCE(SAL,0))                 30/3
--------------------           ----------
                  10                   10
```

COALESCE 函数会返回其参数列表中的第一个非 Null 值。如果是 Null，则 SAL 的值为 0，这样平均值的计算结果也会发生变化。当使用聚合函数的时候，一定要先想一下如何处理 Null。

本解决方案的第二部分使用 GROUP BY（第 3 行）依据部门从属关系把员工数据分为若干组。GROUP BY 使得像 AVG 这样的聚合函数自动为每一个分组返回一个结果。本例中，AVG 函数将针对按照部门分组后的每一组员工数据进行计算。

顺便说一下，不一定非要把 GROUP BY 列放到 SELECT 列表里。来看下面这个例子。

```
select avg(sal)
from emp
group by deptno

AVG(SAL)
----------
2916.66667
      2175
1566.66667
```

尽管 DEPTNO 没有被放在 SELECT 子句里，但是我们仍然按照该列分组。把用于分组的列放入 SELECT 子句通常有助于提高可读性，但这并不是必需的。反之则不然，不能把 GROUP BY 子句里没有的列放入 SELECT 列表。

4. 参考资料

附录 A 能帮你复习与 GROUP BY 相关的知识。

7.2　查找最小值和最大值

1. 问题

你想查找指定列的最大值和最小值。例如，你希望找出全体员工的最高工资和最低工资，以及每个部门的最高工资和最低工资。

2. 解决方案

要查找全体员工的最低工资和最高工资，只需要分别使用 MIN 函数和 MAX 函数即可。

```
1 select min(sal) as min_sal, max(sal) as max_sal
2   from emp

   MIN_SAL    MAX_SAL
---------- ----------
       800       5000
```

为了查找每个部门里的最低工资和最高工资，需要使用 GROUP BY 子句、MIN 函数和 MAX 函数。

```
1 select deptno, min(sal) as min_sal, max(sal) as max_sal
2   from emp
3  group by deptno

    DEPTNO    MIN_SAL    MAX_SAL
---------- ---------- ----------
        10       1300       5000
        20        800       3000
        30        950       2850
```

3. 讨论

为了找到最小值或者最大值，并且把整个表视为一个分组，只需要针对感兴趣的列分别使用 MIN 函数或 MAX 函数即可，不需要使用 GROUP BY 子句。

注意，MIN 函数和 MAX 函数会忽略 Null，而我们可能会遇到 Null 分组，也可能会在一个分组里遇到 Null 值。在下面的查询中，GROUP BY 查询的结果里有两个分组（DEPTNO 分别等于 10 和 30）会返回 Null 值。

```
select deptno, comm
  from emp
 where deptno in (10,30)
 order by 1

    DEPTNO       COMM
---------- ----------
        10
        10
        10
        30        300
```

```
         30        500
         30
         30          0
         30       1300
         30

select min(comm), max(comm)
  from emp

 MIN(COMM)  MAX(COMM)
---------- ----------
         0       1300

select deptno, min(comm), max(comm)
  from emp
 group by deptno

    DEPTNO  MIN(COMM)  MAX(COMM)
---------- ---------- ----------
        10
        20
        30          0       1300
```

正如附录 A 所指出的，即使 SELECT 子句里只有聚合函数，我们仍然能按照表中的其他列分组，如下所示。

```
select min(comm), max(comm)
  from emp
 group by deptno

 MIN(COMM)  MAX(COMM)
---------- ----------
         0       1300
```

尽管 DEPTNO 不在 SELECT 子句里，我们仍然按照它分组。把用于分组的列放入 SELECT 子句通常有助于提高可读性，但这并不是必需的。反之，对于含有 GROUP BY 的查询而言，SELECT 列表里的列必须同时出现在 GROUP BY 子句里。

4. 参考资料

附录 A 能帮你复习与 GROUP BY 相关的知识。

7.3　求和

1. 问题

对所有值求和，例如，计算全体员工的工资总和。

2. 解决方案

针对整个表求和时，只要对感兴趣的列执行 SUM 函数即可，不需要使用 GROUP BY 子句。

```
1 select sum(sal)
2   from emp
```

```
  SUM(SAL)
----------
     29025
```

如果把数据分为多组，就需要使用 SUM 函数和 GROUP BY 子句。下面的例子按照部门计算员工的工资总额。

```
1 select deptno, sum(sal) as total_for_dept
2   from emp
3  group by deptno

    DEPTNO TOTAL_FOR_DEPT
---------- --------------
        10           8750
        20          10875
        30           9400
```

3. 讨论

当计算各部门的工资总额时，其实是在对数据进行分组。对于每个部门而言，所有员工的工资会被相加，并得出总和。本例是一个用 SQL 做聚合运算的例子，因为重点关注的不是某个员工的工资这样的详细信息，而是各个部门的工资总额。注意，SUM 函数会忽略 Null，但是我们可能会遇到 Null 分组，下面的例子会展示这一点。DEPTNO 为 10 的员工都没有业务提成，因而对于 DEPTNO 等于 10 的分组而言，计算 COMM 的总和会返回 Null。

```
select deptno, comm
  from emp
 where deptno in (10,30)
 order by 1

    DEPTNO       COMM
---------- ----------
        10
        10
        10
        30        300
        30        500
        30
        30          0
        30       1300
        30

select sum(comm)
  from emp

 SUM(COMM)
----------
      2100

select deptno, sum(comm)
  from emp
 where deptno in (10,30)
 group by deptno
```

```
    DEPTNO  SUM(COMM)
---------- ----------
        10
        30       2100
```

4. 参考资料

附录 A 能帮你复习与 GROUP BY 相关的知识。

7.4 计算行数

1. 问题

计算一个表的总行数，或者计算某列中值的个数。例如，希望知道员工总数和各部门的员工人数。

2. 解决方案

当计算整个表的总行数时，只需要使用 COUNT 函数和符号 * 即可。

```
1 select count(*)
2   from emp

  COUNT(*)
----------
        14
```

如果把数据分为多组，就需要使用 COUNT 函数和 GROUP BY 子句。

```
1 select deptno, count(*)
2   from emp
3  group by deptno

    DEPTNO   COUNT(*)
---------- ----------
        10          3
        20          5
        30          6
```

3. 讨论

计算各部门的员工人数的时候，我们就是在对数据进行分组。每多一个员工，则总数加一，最终会计算出各个部门的员工人数。本例使用了 SQL 做聚合运算，因为重点关注的不是诸如某个员工的工资或职位这样的详细信息，而是各个部门的员工人数。注意，当把列名称作为参数的时候，COUNT 函数会忽略 Null。使用符号 * 或者常量参数的时候，就会包含 Null，考虑下面的例子。

```
select deptno, comm
  from emp

    DEPTNO       COMM
---------- ----------
        20
        30        300
        30        500
```

```
        20
        30       1300
        30
        10
        20
        10
        30          0
        20
        30
        20
        10
```

```
select count(*), count(deptno), count(comm), count('hello')
  from emp

  COUNT(*) COUNT(DEPTNO) COUNT(COMM) COUNT('HELLO')
---------- ------------- ----------- --------------
        14            14           4             14
```

```
select deptno, count(*), count(comm), count('hello')
  from emp
 group by deptno

    DEPTNO   COUNT(*) COUNT(COMM) COUNT('HELLO')
---------- ---------- ----------- --------------
        10          3           0              3
        20          5           0              5
        30          6           4              6
```

如果参数列的值都为 Null，或者表里没有任何数据，COUNT 函数就会返回 0。请注意，即使 SELECT 子句里只有聚合函数，我们仍然可以按照其他列对表中的数据进行分组，如下所示。

```
select count(*)
  from emp
 group by deptno

  COUNT(*)
----------
         3
         5
         6
```

注意，尽管 DEPTNO 不存在于 SELECT 列表里，我们仍然按照该列进行分组。把用于分组的列放入 SELECT 子句通常有助于提高可读性，但这并不是必需的。反之，如果某列被放入了 SELECT 列表，那么我们就必须保证它也存在于 GROUP BY 子句里。

4. 参考资料

附录 A 能帮你复习与 GROUP BY 相关的知识。

7.5　计算非Null值的个数

1. 问题

计算某一列里非 Null 值的个数。例如，希望知道有多少名员工获得了业务提成。

2. 解决方案

计算 EMP 表的 COMM 列里非 Null 值的个数。

```
select count(comm)
  from emp

COUNT(COMM)
-----------
          4
```

3. 讨论

当执行 COUNT(*) 操作时，实际上是在统计行数（而不会去管实际的值是什么，这就是为什么 Null 值和非 Null 值都会被计入总数）。但是，如果针对某一列执行 COUNT 操作，我们却是在计算该列非 Null 值的个数。前一个实例的“讨论”部分提到了这种区别。在本解决方案中，COUNT(COMM) 返回的是 COMM 列非 Null 值的个数。因为只有获得业务提成的员工才会在对应的 COMM 列有非 Null 值，所以 COUNT(COMM) 的计数结果就是这一类员工的总数。

7.6　累计求和

1. 问题

你想针对某一列进行累计求和。

2. 解决方案

作为示例，下面的解决方案将介绍如何计算全体员工工资的累计额。为了便于理解，查询结果按照 SAL 列排序，这样很容易观察累计和的增长。

DB2 和 Oracle

使用 SUM 函数的窗口函数版本进行累计求和。

```
1 select ename, sal,
2        sum(sal) over (order by sal,empno) as running_total
3   from emp
4   order by 2

ENAME             SAL RUNNING_TOTAL
---------- ---------- -------------
SMITH             800           800
JAMES             950          1750
ADAMS            1100          2850
WARD             1250          4100
MARTIN           1250          5350
MILLER           1300          6650
TURNER           1500          8150
```

```
ALLEN            1600          9750
CLARK            2450         12200
BLAKE            2850         15050
JONES            2975         18025
SCOTT            3000         21025
FORD             3000         24025
KING             5000         29025
```

MySQL、PostgreSQL 和 SQL Server

使用标量子查询来进行累计求和（而不是使用如 SUM OVER 这样的窗口函数，因为不能像 DB2 和 Oracle 的解决方案那样方便地按照 SAL 列对结果集排序）。最终的累计和是正确的（最后一行的合计值与前面的 DB2 和 Oracle 解决方案的结果一致），但是中间结果因为缺少了排序操作而不尽相同。

```
1 select e.ename, e.sal,
2        (select sum(d.sal) from emp d
3          where d.empno <= e.empno) as running_total
4   from emp e
5  order by 3

ENAME             SAL RUNNING_TOTAL
---------- ---------- -------------
SMITH             800           800
ALLEN            1600          2400
WARD             1250          3650
JONES            2975          6625
MARTIN           1250          7875
BLAKE            2850         10725
CLARK            2450         13175
SCOTT            3000         16175
KING             5000         21175
TURNER           1500         22675
ADAMS            1100         23775
JAMES             950         24725
FORD             3000         27725
MILLER           1300         29025
```

3. 讨论

使用新的 ANSI 窗口函数很容易进行累计求和。对于尚未支持窗口函数的数据库管理系统而言，必须使用标量子查询（通过一个具有唯一值的字段做连接查询）。

DB2 和 Oracle

使用窗口函数 SUM OVER 很容易进行累计求和。上述解决方案使用的 ORDER BY 子句的后面不仅有 SAL 列，还有 EMPNO 列（该列是主键），这是为了防止累计求和过程中出现重复值。以下示例中的 RUNNING_TOTAL2 列展示了重复值会导致什么样的问题。

```
select empno, sal,
       sum(sal)over(order by sal,empno) as running_total1,
       sum(sal)over(order by sal) as running_total2
  from emp
 order by 2
```

```
ENAME             SAL RUNNING_TOTAL1 RUNNING_TOTAL2
---------- ---------- -------------- --------------
SMITH             800            800            800
JAMES             950           1750           1750
ADAMS            1100           2850           2850
WARD             1250           4100           5350
MARTIN           1250           5350           5350
MILLER           1300           6650           6650
TURNER           1500           8150           8150
ALLEN            1600           9750           9750
CLARK            2450          12200          12200
BLAKE            2850          15050          15050
JONES            2975          18025          18025
SCOTT            3000          21025          24025
FORD             3000          24025          24025
KING             5000          29025          29025
```

员工 WARD、MARTIN、SCOTT 和 FORD 对应的 RUNNING_TOTAL2 是不正确的。这是因为他们的工资在 EMP 表里出现了不止一次，而这些重复值也被计算到累计和中。这就是为什么必须把 EMPNO 列（它是唯一的）加入排序项，才能得到正确的计算结果，即 RUNNING_TOTAL1。考虑这样一种情况：对于 ADAMS，RUNNING_TOTAL1 和 RUNNING_TOTAL2 两列都是 2850。2850 加上 WARD 的工资 1250，应该得到 4100，然而 RUNNING_TOTAL2 返回的却是 5350。这是为什么呢？因为 WARD 和 MARTIN 的工资相同，这两个 1250 相加得到 2500，再加上 2850 就是 WARD 和 MARTIN 这两行对应的 5350。通过把若干列组合起来作为排序列，能够避免出现重复值（例如，SAL 和 EMPNO 的组合是唯一的），这样就能保证得到正确的累计和。

MySQL、PostgreSQL 和 SQL Server

因为这些数据库管理系统尚未完全支持窗口函数，所以我们只能通过标量子查询来计算累计和。必须通过一个有唯一值的列做连接查询，否则，如果两个员工的工资相同，那么得到的累计和就是错误的。对于本实例而言，关键在于把 D.EMPNO 和 E.EMPNO 连接起来，并针对每个小于或者等于 E.EMPNO 的 D.EMPNO 计算出对应的 D.SAL。如果把标量子查询改写成一个 EMP 表全体数据与部分数据之间的连接查询，会更加容易理解这一点。

```
select e.ename as ename1, e.empno as empno1, e.sal as sal1,
       d.ename as ename2, d.empno as empno2, d.sal as sal2
  from emp e, emp d
 where d.empno <= e.empno
   and e.empno = 7566

ENAME          EMPNO1       SAL1 ENAME          EMPNO2       SAL2
---------- ---------- ---------- ---------- ---------- ----------
JONES            7566       2975 SMITH            7369        800
JONES            7566       2975 ALLEN            7499       1600
JONES            7566       2975 WARD             7521       1250
JONES            7566       2975 JONES            7566       2975
```

EMPNO2 列中的每一个值都会和 EMPNO1 列中对应的值进行比较。如果 EMPNO2 列中的值小于或者等于 EMPNO1 列中对应的值，则其对应的 SAL2 列的值会被计入总和。对于上述查询而

言，员工 SMITH、ALLEN、WARD 和 JONES 的 EMPNO 值会和 JONES 的 EMPNO 值相比较。由于四个员工的 EMPNO 值都满足小于或者等于 JONES 的 EMPNO 值这一条件，因此他们的工资都会被计入总和。反之，（在上述查询中）任何 EMPNO 值大于 JONES 的 EMPNO 值的员工，其工资都不会被计入总和。对于完整的查询而言，其工作方式也是同样的，如果 EMPNO 值小于或者等于全表的最大值 7934（MILLER 的 EMPNO 值），则其对应的工资就会被计入总和。

7.7 计算累计乘积

1. 问题

你想计算某个数值列的累计乘积。这个操作类似于上一个实例，只不过不使用加法，而改用乘法。

2. 解决方案

例如，你想计算员工工资的累计乘积。虽然工资的累计乘积可能用处不大，但同样的方法也能方便地应用于其他更有用的领域。

DB2 和 Oracle

使用窗口函数 SUM OVER，并利用对数来模拟乘法。

```
1 select empno,ename,sal,
2        exp(sum(ln(sal))over(order by sal,empno)) as running_prod
3   from emp
4  where deptno = 10

EMPNO ENAME       SAL         RUNNING_PROD
----- ---------- ---- --------------------
 7934 MILLER     1300                 1300
 7782 CLARK      2450              3185000
 7839 KING       5000          15925000000
```

不能使用 SQL 计算负数和零的对数。如果表里有这样的值，应该避免把它们传递给 SQL 的 LN 函数。为了便于理解，本解决方案并没有针对这些值和 Null 做防范处理，但是在真实环境下，你应该考虑在代码中添加防范措施。如果你一定要处理负数和零，那么该解决方案可能不适用。

还有一个只适用于 Oracle 的解决方案，那就是使用 Oracle Database 10*g* 开始支持的 MODEL 子句。在下面的例子里，SAL 列中的每一个值都会被转换成负数，这是为了证明负数不会给计算累计乘积带来任何问题。

```
1 select empno, ename, sal, tmp as running_prod
2   from (
3 select empno,ename,-sal as sal
4   from emp
5  where deptno=10
6        )
7  model
8    dimension by(row_number() over(order by sal desc) rn )
```

```
 9    measures(sal, 0 tmp, empno, ename)
10    rules (
11      tmp[any] = case when sal[cv()-1] is null then sal[cv()]
12                      else tmp[cv()-1]*sal[cv()]
13                 end
14    )

EMPNO ENAME       SAL         RUNNING_PROD
----- ---------- ---- --------------------
 7934 MILLER    -1300                -1300
 7782 CLARK     -2450              3185000
 7839 KING      -5000         -15925000000
```

MySQL、PostgreSQL 和 SQL Server

我们仍然需要使用对数求和，但是这些数据库不支持窗口函数，因而改用标量子查询。

```
1 select e.empno,e.ename,e.sal,
2        (select exp(sum(ln(d.sal)))
3           from emp d
4          where d.empno <= e.empno
5            and e.deptno=d.deptno) as running_prod
6   from emp e
7   where e.deptno=10

EMPNO ENAME       SAL         RUNNING_PROD
----- ---------- ---- --------------------
 7782 CLARK      2450                 2450
 7839 KING       5000             12250000
 7934 MILLER     1300          15925000000
```

对于 SQL Server 而言，还需要用 `LOG` 函数来替代 `LN` 函数。

3. 讨论

除了仅适用于 Oracle Database 10*g* 及其后续版本的 `MODEL` 子句，另外两个解决方案都使用了下面的把两个数字累加的做法：

(1) 计算它们各自的自然对数；
(2) 把上述自然对数的计算结果累加起来；
(3) 把上述累加结果作为指数，以数学常量 *e* 为底数，进行幂运算（使用 `EXP` 函数）。

注意，上述做法不适用于零和负数的累计，因为 SQL 的对数函数不支持小于或等于零的值。

DB2 和 Oracle

关于窗口函数 `SUM OVER` 的工作原理，请参考前一个实例的相关内容。

对于 Oracle Database 10*g* 及其后续版本，可以使用 `MODEL` 子句来计算累计乘积。使用 `MODEL` 子句和窗口函数 `ROW_NUMBER`，很容易找到当前行的前一行。可以像访问数组一样访问 `MEASURES` 列表的每一项，也可以使用 `DIMENSIONS` 列表的项（即 `ROW_NUMBER` 函数的返回值，别名为 `RN`）来查找数组。

```
select empno, ename, sal, tmp as running_prod,rn
  from (
select empno,ename,-sal as sal
  from emp
 where deptno=10
       )
 model
   dimension by(row_number() over(order by sal desc) rn )
   measures(sal, 0 tmp, empno, ename)
  rules ()

EMPNO ENAME             SAL RUNNING_PROD         RN
----- ---------- ---------- ------------ ----------
 7934 MILLER          -1300            0          1
 7782 CLARK           -2450            0          2
 7839 KING            -5000            0          3
```

我们看到 SAL[1] 的值是 -1300。由于数字从 1 开始连续增加，因此能通过把行号减去一来访问前一行。RULES 子句如下所示。

```
rules (
   tmp[any] = case when sal[cv()-1] is null then sal[cv()]
                   else tmp[cv()-1]*sal[cv()]
              end
 )
```

使用内置运算符 ANY 可以遍历每一行，而无须硬编码。在这个例子里，ANY 会被分别赋值为 1、2 和 3。TMP[n] 的初始值为 0。通过评估当前的值（CV 函数返回当前值）来决定 TMP[n] 的值。TMP[1] 的初始值为 0，而 SAL[1] 是 -1300。因为 SAL[0] 不存在，所以 TMP[1] 被设置为 SAL[1]。设置好 TMP[1] 后，下一行是 TMP[2]。SAL[1] 会被评估（SAL[CV()-1] 是 SAL[1]，因为 ANY 的当前值是 2）。SAL[1] 不是 Null，它的值为 -1300，因而 TMP[2] 就是 TMP[1] 和 SAL[2] 的乘积。以此类推，直至完成所有行的计算。

MySQL、PostgreSQL 和 SQL Server

请参考前一个实例里针对 MySQL、PostgreSQL 和 SQL Server 解决方案的标量子查询的解释。

注意，基于子查询的解决方案得到的输出结果与 Oracle 和 DB2 解决方案的略有不同，这是由于多了针对 EMPNO 列的比较运算（累计乘积以一种不同的顺序被计算出来）。类似于累计求和的做法，标量子查询驱动了乘积的累计；基于子查询的解决方案按照 EMPNO 列对数据进行排序，而在 Oracle 和 DB2 解决方案里则按照 SAL 列排序。

7.8 计算累计差

1. 问题

计算某个数值列的累计差。例如，希望计算 DEPTNO 等于 10 的部门里员工工资的累计差，并且得到如下所示的结果集。

```
ENAME             SAL RUNNING_DIFF
---------- ---------- ------------
MILLER           1300         1300
CLARK            2450        -1150
KING             5000        -6150
```

2. 解决方案

DB2 和 Oracle

使用窗口函数 SUM OVER 计算累计差。

```
1  select ename,sal,
2         sum(case when rn = 1 then sal else -sal end)
3         over(order by sal,empno) as running_diff
4    from (
5  select empno,ename,sal,
6         row_number() over(order by sal,empno) as rn
7    from emp
8   where deptno = 10
9         ) x
```

MySQL、PostgreSQL 和 SQL Server

使用标量子查询计算累计差。

```
1 select a.empno, a.ename, a.sal,
2        (select case when a.empno = min(b.empno) then sum(b.sal)
3                     else sum(-b.sal)
4                 end
5           from emp b
6          where b.empno <= a.empno
7            and b.deptno = a.deptno ) as rnk
8   from emp a
9  where a.deptno = 10
```

3. 讨论

本实例的解决方案和 7.6 节的类似。唯一的不同之处在于 SAL 的结果都是负数，当然第一行除外（本例把 DEPTNO 等于 10 的第一个 SAL 值作为起点）。

7.9 计算众数

1. 问题

需要找出某一列的众数（即在一组数据里出现次数最多的那个数）。例如，希望找出 DEPTNO 等于 20 的部门里员工工资的众数。就如下示例而言，众数为 3000。

```
select sal
  from emp
 where deptno = 20
 order by sal

       SAL
----------
       800
      1100
```

```
2975
3000
3000
```

2. 解决方案

DB2 和 SQL Server

使用窗口函数 DENSE_RANK 对工资值出现的次数进行排序，以帮助我们找到众数。

```
 1  select sal
 2    from (
 3  select sal,
 4         dense_rank() over(order by cnt desc) as rnk
 5    from (
 6  select sal, count(*) as cnt
 8    from emp
 9   where deptno = 20
10   group by sal
11         ) x
12         ) y
13   where rnk = 1
```

Oracle

对于 Oracle 8*i* 来说，可以使用 DB2 的解决方案。如果你使用的是 Oracle 9*i* 及其后续版本，可以使用聚合函数 MAX 的 KEEP 扩展来找到 SAL 列的众数。注意，如果有多个众数，则 KEEP 解决方案只会保留工资值最高的那个。如果希望看到全部众数，就需要修改该解决方案，或者改用前面的 DB2 解决方案。在本例中，因为 3000 是 DEPTNO 等于 20 的部门里 SAL 列的众数，并且也是 SAL 列的最大值，所以下面的解决方案足以胜任。

```
1  select max(sal)
2          keep(dense_rank first order by cnt desc) sal
3    from (
4  select sal, count(*) cnt
5    from emp
6   where deptno=20
7   group by sal
8         )
```

MySQL 和 PostgreSQL

使用子查询查找众数。

```
1  select sal
2    from emp
3   where deptno = 20
4   group by sal
5  having count(*) >= all ( select count(*)
6                             from emp
7                            where deptno = 20
8                            group by sal )
```

3. 讨论

DB2 和 SQL Server

内嵌视图 X 返回每一个 SAL 值及其出现的次数。内嵌视图 Y 使用窗口函数 DENSE_RANK（该

函数允许出现排名相同的状况）对查询结果进行排序。

查询结果按照每一个 SAL 值的出现次数进行排序，如下所示。

```
1 select sal,
2        dense_rank()over(order by cnt desc) as rnk
3   from (
4 select sal,count(*) as cnt
5   from emp
6  where deptno = 20
7  group by sal
8        ) x

  SAL        RNK
----- ----------
 3000          1
  800          2
 1100          2
 2975          2
```

最外层的查询只是简单地筛选出 RNK 等于 1 的行。

Oracle

内嵌视图返回每一个 SAL 值及其出现的次数，如下所示。

```
select sal, count(*) cnt
  from emp
 where deptno=20
 group by sal

  SAL        CNT
----- ----------
  800          1
 1100          1
 2975          1
 3000          2
```

然后利用聚合函数 MAX 的 KEEP 扩展找到众数。下面的 KEEP 子句包含 DENSE_RANK、FIRST 和 ORDER BY CNT DESC 三个部分。

```
keep(dense_rank first order by cnt desc)
```

这样做很容易查找众数。KEEP 子句会查看内嵌视图返回的 CNT 值，并决定哪一个 SAL 值会被 MAX 函数返回。按照从右向左的执行顺序，所有的 CNT 值先按照降序排列，然后执行 DENSE_RANK 函数，并保留排在第一位的 CNT 值。仔细观察内嵌视图查询结果，可以发现 3000 对应的 CNT 是 2，在查询结果中是最大的。MAX(SAL) 的返回值是最大的 SAL 值，而且其对应的 CNT 值也是最大的，本例中是 3000。

4. 参考资料

11.11 节对 Oracle 聚合函数的 KEEP 扩展有更深入的讨论。

MySQL 和 PostgreSQL

子查询返回每一个 SAL 值的出现次数。外层查询返回出现次数大于或者等于子查询全部

返回值的 SAL 值（也就是说，外层查询返回 DEPTNO 等于 20 的部门里出现次数最多的工资值）。

7.10　计算中位数

1. 问题

你想计算某个数值列的中位数（即按顺序排列的一组数据中居于中间位置的数）。例如，你希望知道 DEPTNO 等于 20 的部门里员工工资的中位数。就如下示例而言，中位数为 2975。

```
select sal
  from emp
 where deptno = 20
 order by sal

       SAL
----------
       800
      1100
      2975
      3000
      3000
```

2. 解决方案

除了 Oracle 的解决方案（Oracle 有可以计算中位数的函数），其他解决方案都基于 David Rozenshtein、Anatoly Abramovich 和 Eugene Birger 在 *Optimizing Transact-SQL* 一书中记载的方法。窗口函数的出现使得我们有了比传统的自连接查询更为高效的做法。

DB2

使用窗口函数 COUNT(*) OVER 和 ROW_NUMBER 查找中位数。

```
 1 select avg(sal)
 2   from (
 3 select sal,
 4        count(*) over() total,
 5        cast(count(*) over() as decimal)/2 mid,
 6        ceil(cast(count(*) over() as decimal)/2) next,
 7        row_number() over (order by sal) rn
 8   from emp
 9  where deptno = 20
10        ) x
11  where ( mod(total,2) = 0
12          and rn in ( mid, mid+1 )
13        )
14     or ( mod(total,2) = 1
15          and rn = next
16        )
```

MySQL 和 PostgreSQL

使用自连接查询查找中位数。

```
1  select avg(sal)
2    from (
3  select e.sal
4    from emp e, emp d
5   where e.deptno = d.deptno
6     and e.deptno = 20
7   group by e.sal
8  having sum(case when e.sal = d.sal then 1 else 0 end)
9                            >= abs(sum(sign(e.sal - d.sal)))
10         )
```

Oracle

使用函数 `MEDIAN`（Oracle Database 10*g*）或者 `PERCENTILE_CONT`（Oracle 9*i*）。

```
1 select median(sal)
2   from emp
3  where deptno=20

1 select percentile_cont(0.5)
2         within group(order by sal)
3   from emp
4  where deptno=20
```

DB2 解决方案也适用于 Oracle 8*i*。对于 Oracle 8*i* 之前的版本，就要采用 MySQL 和 PostgreSQL 的解决方案。

SQL Server

使用窗口函数 `COUNT(*) OVER` 和 `ROW_NUMBER` 查找中位数。

```
1  select avg(sal)
2    from (
3  select sal,
4         count(*) over() total,
5         cast(count(*) over() as decimal)/2 mid,
6         ceiling(cast(count(*)over() as decimal)/2) next,
7         row_number() over(order by sal) rn
8    from emp
9   where deptno = 20
10         ) x
11   where ( total%2 = 0
12           and rn in ( mid, mid+1 )
13         )
14      or ( total%2 = 1
15           and rn = next
16         )
```

3. 讨论

DB2 和 SQL Server

DB2 和 SQL Server 的解决方案之间的唯一不同之处在于一个语法细节：SQL Server 的取模运算符是 `%`，DB2 则使用 `MOD` 函数。除此之外，二者并无二致。内嵌视图 `X` 返回 `TOTAL`、`MID` 和 `NEXT` 三种不同的总数以及 `ROW_NUMBER` 函数生成的 `RN`。这些额外生成的列有助于查找中位数。仔细观察内嵌视图 `X` 返回的结果集，并看看这些列表示什么。

```
select sal,
       count(*) over() total,
       cast(count(*)over() as decimal)/2 mid,
       ceil(cast(count(*)over() as decimal)/2) next,
       row_number()over(order by sal) rn
  from emp
 where deptno = 20

 SAL TOTAL  MID NEXT   RN
---- ----- ---- ---- ----
 800     5  2.5    3    1
1100     5  2.5    3    2
2975     5  2.5    3    3
3000     5  2.5    3    4
3000     5  2.5    3    5
```

为了找到中位数，SAL 列的值必须按照从小到大的顺序排列。由于 DEPTNO 等于 20 的员工的个数为奇数，因此中位数就是 RN 与 NEXT 相等的那一行的 SAL 值（NEXT 表示大于员工总数除以 2 的商的最小整数）。

如果内嵌视图 X 返回的结果集的行数为奇数，则 WHERE 子句的前半部分（第 11 ～ 13 行）不会命中。如果我们确信内嵌视图 X 返回的结果集的行数始终是奇数，不妨简化为如下的形式。

```
select avg(sal)
  from (
select sal,
       count(*)over() total,
       ceil(cast(count(*)over() as decimal)/2) next,
       row_number()over(order by sal) rn
  from emp
 where deptno = 20
       ) x
 where rn = next
```

不幸的是，如果子查询返回的结果集的行数为偶数，上述简化的做法不再适用。该解决方案使用 MID 列来处理偶数行的情况。如果 DEPTNO 等于 30，则内嵌视图 X 的返回结果会包含 6 个员工。

```
select sal,
       count(*)over() total,
       cast(count(*)over() as decimal)/2 mid,
       ceil(cast(count(*)over() as decimal)/2) next,
       row_number()over(order by sal) rn
  from emp
 where deptno = 30

 SAL TOTAL  MID NEXT   RN
---- ----- ---- ---- ----
 950     6    3    3    1
1250     6    3    3    2
1250     6    3    3    3
1500     6    3    3    4
1600     6    3    3    5
2850     6    3    3    6
```

由于一共有偶数行数据，因此中位数就变成了其中两行数据的平均值，相关的两行是 RN 等于 MID 的那一行以及 RN 等于 MID+1 的那一行。

MySQL 和 PostgreSQL

首先通过自连接 EMP 表来计算中位数，自连接查询会返回所有工资值的笛卡儿积（但是针对 E.SAL 做 GROUP BY 之后会去掉重复值）。HAVING 子句使用 SUM 函数来计算 E.SAL 和 D.SAL 相等的次数。如果它们相等的次数不小于 E.SAL 大于 D.SAL 的次数，则对应的行就是中位数。不妨把 SUM 函数添加到 SELECT 列表中，以观察查询结果，并验证这一点。

```
select e.sal,
       sum(case when e.sal=d.sal
                then 1 else 0 end) as cnt1,
       abs(sum(sign(e.sal - d.sal))) as cnt2
  from emp e, emp d
 where e.deptno = d.deptno
   and e.deptno = 20
 group by e.sal

 SAL CNT1 CNT2
---- ---- ----
 800    1    4
1100    1    2
2975    1    0
3000    4    6
```

Oracle

对于 Oracle Database 10*g* 或者 Oracle 9*i*，可以使用 Oracle 中的函数计算中位数。对于 Oracle 8*i*，不妨使用 DB2 解决方案。对于更早的 Oracle 版本，只能采用 PostgreSQL 解决方案。MEDIAN 函数明显是用于计算中位数的，而 PERCENTILE_CONT 函数的用途看起来就不那么直观。传递给 PERCENTILE_CONT 函数的参数 0.5 其实是一个百分位数值。WITHIN GROUP (ORDER BY SAL) 子句会生成顺序排列的行数据，以便于 PERCENTILE_CONT 函数搜索。（注意，中位数就是在一组顺序排列的数值里居于中间位置的那个数。）最后的返回值是顺序排列的行数据里落入指定百分位的数值（本例中的百分位数值是 0.5，居于边界值 0 和 1 的中间）。

7.11　计算百分比

1. 问题

你想知道某一列的值占总和的百分比。例如，你希望知道 DEPTNO 等于 10 的部门的工资占全体员工工资的百分比。

2. 解决方案

总体而言，使用 SQL 计算百分比和在纸上手算没什么不同。只需先做除法，再做乘法即可。在本例中，要计算 EMP 表中 DEPTNO 等于 10 的工资额占总体的百分比。先算出 DEPTNO 等于 10 的工资总额，然后再除以表中全部工资的总额，最后乘以 100 以得到一个代表百分比的值。

MySQL 和 PostgreSQL

DEPTNO 等于 10 的工资总额除以全体工资总额。

```
1 select (sum(
2          case when deptno = 10 then sal end)/sum(sal)
3         )*100 as pct
4   from emp
```

DB2、Oracle 和 SQL Server

使用内嵌视图和窗口函数 SUM OVER 来得到全体工资总额以及 DEPTNO 等于 10 的工资总额。然后，在外层查询中执行除法和乘法。

```
1  select distinct (d10/total)*100 as pct
2    from (
3  select deptno,
4         sum(sal)over() total,
5         sum(sal)over(partition by deptno) d10
6    from emp
7         ) x
8   where deptno=10
```

3. 讨论

MySQL 和 PostgreSQL

CASE 语句很容易筛选出 DEPTNO 等于 10 的工资值，把这些数加起来就可以得到工资和，然后除以工资总和。因为聚合函数忽略 Null，所以不需要在 CASE 语句的后面加 ELSE 子句。为了清楚地看到除数和被除数，不妨先去掉除法运算，并执行如下的查询语句。

```
select sum(case when deptno = 10 then sal end) as d10,
       sum(sal)
  from emp

 D10  SUM(SAL)
---- ---------
8750     29025
```

执行除法运算时可能需要加入显式的类型转换操作，这取决于 SAL 列的类型。例如，对于 DB2、SQL Server 和 PostgreSQL，如果 SAL 列的类型为整型，可以将其转换为十进制小数，以便于得到正确的计算结果，如下所示。

```
select (cast(
         sum(case when deptno = 10 then sal end)
             as decimal)/sum(sal)
        )*100 as pct
  from emp
```

DB2、Oracle 和 SQL Server

与上述传统的做法不同，下面的解决方案使用窗口函数来计算百分比。对于 DB2 和 SQL Server，如果 SAL 列的类型为整型，需要在进行除法运算前做类型转换。

```
select distinct
       cast(d10 as decimal)/total*100 as pct
  from (
select deptno,
```

```
       sum(sal)over() total,
       sum(sal)over(partition by deptno) d10
  from emp
       ) x
 where deptno=10
```

始终要记住的一点是，WHERE 子句评估完成之后才会执行窗口函数。因而，不能将过滤 DEPTNO 的操作放到内嵌视图 X 里。试想没有 DEPTNO 过滤条件和有该过滤条件，内嵌视图 X 的查询结果集有何异同。首先来看看没有该过滤条件的结果集。

```
select deptno,
       sum(sal)over() total,
       sum(sal)over(partition by deptno) d10
  from emp

 DEPTNO     TOTAL       D10
------- --------- ---------
     10     29025      8750
     10     29025      8750
     10     29025      8750
     20     29025     10875
     20     29025     10875
     20     29025     10875
     20     29025     10875
     20     29025     10875
     30     29025      9400
     30     29025      9400
     30     29025      9400
     30     29025      9400
     30     29025      9400
     30     29025      9400
```

下面是有过滤条件的结果集。

```
select deptno,
       sum(sal)over() total,
       sum(sal)over(partition by deptno) d10
  from emp
 where deptno=10

DEPTNO     TOTAL       D10
------ --------- ---------
    10      8750      8750
    10      8750      8750
    10      8750      8750
```

因为要先评估 WHERE 子句再执行窗口函数，所以此处 TOTAL 的计算结果实际上只是 DEPTNO 等于 10 的员工的工资和。但是，我们希望 TOTAL 等于全体员工的工资和。这就是为什么 DEPTNO 的过滤条件要放在内嵌视图 X 之外。

7.12 聚合Null列

1. 问题

你想针对某列做聚合运算，但该列的值为 Null。你希望保持聚合运算结果的准确性，但又

担心聚合函数会忽略 Null 值。例如，你想知道 DEPTNO 等于 30 的员工的平均业务提成，但部分员工实际上没有获得提成（这些人的 COMM 列是 Null）。由于在聚合运算过程中 Null 会被忽略，因此输出结果的准确性就无法得到保证。总之，你希望聚合运算能以某种方式把 Null 值也包含进去。

2. 解决方案

使用 COALESCE 函数把 Null 转换为 0，这样聚合函数就能处理它们了。

```
1  select avg(coalesce(comm,0)) as avg_comm
2    from emp
3   where deptno=30
```

3. 讨论

使用聚合函数时一定要记住，Null 值会被忽略。来看看不使用 COALESCE 函数的输出结果。

```
select avg(comm)
  from emp
 where deptno=30

AVG(COMM)
---------
      550
```

上述查询结果显示，DEPTNO 等于 30 的员工的平均业务提成是 550，但如果快速查看一下，就会发现实际情形并非如此。

```
select ename, comm
  from emp
 where deptno=30
order by comm desc

ENAME           COMM
---------- ---------
BLAKE
JAMES
MARTIN          1400
WARD             500
ALLEN            300
TURNER             0
```

以上结果集显示，在 6 个员工中，只有 4 人能领取业务提成。DEPTNO 等于 30 的员工的提成总额为 2200，因而平均值应该是 2200/6，而不是 2200/4。如果不使用 COALESCE 函数，我们实际上是在回答这样一个问题："对于 DEPTNO 等于 30、且能领取业务提成的员工而言，其提成平均值是多少？"但实际上我们要回答的问题却是："对于 DEPTNO 等于 30 的全体员工而言，其提成平均值是多少？"总之，一定要记住的是，一旦涉及聚合运算，就要相应地考虑如何处理 Null 值。

7.13 计算平均值时去掉最大值和最小值

1. 问题

你想计算平均值，但又想去掉最大值和最小值，以降低它们对最终计算结果的影响。例

如，你希望先去掉最高工资和最低工资后，再计算全体员工的平均工资。

2. 解决方案

MySQL 和 PostgreSQL

使用子查询去掉最大值和最小值。

```
1  select avg(sal)
2    from emp
3   where sal not in (
4      (select min(sal) from emp),
5      (select max(sal) from emp)
6   )
```

DB2、Oracle 和 SQL Server

使用内嵌视图以及窗口函数 MAX OVER 和 MIN OVER 来生成结果集，可以很容易去掉最大值和最小值。

```
1  select avg(sal)
2    from (
3  select sal, min(sal) over()min_sal, max(sal)over() max_sal
4    from emp
5         ) x
6   where sal not in (min_sal,max_sal)
```

3. 讨论

MySQL 和 PostgreSQL

子查询能找到最高和最低的工资值。针对它们使用 NOT IN 之后，在计算平均值时就能去掉这些值。注意，如果有重复值（例如，最高或者最低工资对应的员工不止一人），那么所有重复值都将被过滤掉。如果希望只去掉一个最大值和一个最小值，只需要把它们从合计值里先减掉，再做除法即可。

```
select (sum(sal)-min(sal)-max(sal))/(count(*)-2)
  from emp
```

DB2、Oracle 和 SQL Server

内嵌视图 X 会返回全部工资值以及最高和最低的工资值。

```
select sal, min(sal)over() min_sal, max(sal)over() max_sal
  from emp

      SAL   MIN_SAL   MAX_SAL
--------- --------- ---------
      800       800      5000
     1600       800      5000
     1250       800      5000
     2975       800      5000
     1250       800      5000
     2850       800      5000
     2450       800      5000
     3000       800      5000
     5000       800      5000
     1500       800      5000
```

```
   1100       800      5000
    950       800      5000
   3000       800      5000
   1300       800      5000
```

在以上的查询结果中，每一行都包含最高和最低的工资值，因而查找最大值和最小值就不是问题。外层查询针对内嵌视图 X 返回的行又做了一次过滤，这样一来，计算平均值之前就先去掉了与 MIN_SAL 或者 MAX_SAL 相等的工资值。

7.14 将含有字母和数字的字符串转换为数字

1. 问题

你有字母和数字混合的数据，并希望提取出其中的数字部分。例如，对于字符串 paul123f321，你想得到的结果是数字 123321。

2. 解决方案

DB2

使用函数 TRANSLATE 和 REPLACE 从含有字母和数字的字符串里提取出数字字符。

```
1 select cast(
2        replace(
3       translate( 'paul123f321',
4                  repeat('#',26),
5                  'abcdefghijklmnopqrstuvwxyz'),'#','')
6         as integer ) as num
7   from t1
```

Oracle 和 PostgreSQL

使用函数 TRANSLATE 和 REPLACE 从含有字母和数字的字符串里提取出数字字符。

```
1 select cast(
2        replace(
3       translate( 'paul123f321',
4                 'abcdefghijklmnopqrstuvwxyz',
5                 rpad('#',26,'#')),'#','')
6        as integer ) as num
7   from t1
```

MySQL 和 SQL Server

在写作本书时，这两个数据库尚未支持 TRANSLATE 函数，因而无法提供解决方案。

3. 讨论

两种解决方案的唯一区别是语法：DB2 使用的是 REPEAT 函数而非 RPAD 函数，并且 TRANSLATE 函数的参数列表顺序也有所不同。下面的解释内容以 Oracle 和 PostgreSQL 解决方案为准，也会适当兼顾 DB2 的解决方案。如果我们试着从内到外执行查询（从 TRANSLATE 开始），就会明白整个查询其实不难。首先，TRANSLATE 函数把每个非数字字符替换为 #。

```
select translate( 'paul123f321',
                  'abcdefghijklmnopqrstuvwxyz',
                  rpad('#',26,'#')) as num
  from t1

NUM
-----------
####123#321
```

接着，使用 REPLACE 函数删除 #，最后再把剩余的部分转换为数值类型即可。这个例子非常简单，因为要处理的数据是一个只包含字母和数字的字符串。如果该字符串还包含其他字符，那么从反面着手可能会更容易：不是先找到非数字字符并删除它们，而是先找到数字字符再去掉其他字符。下面的例子展示了这种方法。

```
select replace(
       translate('paul123f321',
       replace(translate( 'paul123f321',
                          '0123456789',
                          rpad('#',10,'#')),'#',''),
               rpad('#',length('paul123f321'),'#')),'#','') as num
  from t1

NUM
------
123321
```

上述解决方案看起来比最初的做法更令人费解，不过如果拆开来看，也不是很难懂。我们来观察最内层的 TRANSLATE 函数调用。

```
select translate( 'paul123f321',
                  '0123456789',
                  rpad('#',10,'#'))
  from t1

TRANSLATE('
-----------
paul###f###
```

第一步就不同：它没有用 # 替换非数字字符，反而用 # 逐个替换全部数字字符。下一步删除 #，这样就只剩下非数字字符了。

```
select replace(translate( 'paul123f321',
                          '0123456789',
                          rpad('#',10,'#')),'#','')
  from t1

REPLA
-----
paulf
```

然后再次调用 TRANSLATE 函数，用 # 替换原字符串里的所有非数字字符（上一步的返回结果）。

```
select translate('paul123f321',
       replace(translate( 'paul123f321',
                          '0123456789',
                          rpad('#',10,'#')),'#',''),
               rpad('#',length('paul123f321'),'#'))
  from t1

TRANSLATE('
-----------
####123#321
```

这时我们不妨停下来，仔细观察最外层的 TRANSLATE 函数调用。RPAD 函数的第二个参数（对于 DB2 而言，是 REPEAT 函数的第二个参数）是原字符串的长度。这是一种巧妙的用法，因为一个字符串里任何字符的出现次数都不会超过整个字符串的长度。现在，全部非数字字符都被替换成了 #。最后，调用 REPLACE 函数去掉这些 #，因此最终就只剩下数字。

7.15 修改累计值

1. 问题

你想依据另一列的值来修改累计值。试想这样的场景：你希望显示一个信用卡账户的交易历史，并显示每一笔交易完成后的余额。本实例将会用到如下所示的视图 V。

```
create view V (id,amt,trx)
as
select 1, 100, 'PR' from t1 union all
select 2, 100, 'PR' from t1 union all
select 3, 50,  'PY' from t1 union all
select 4, 100, 'PR' from t1 union all
select 5, 200, 'PY' from t1 union all
select 6, 50,  'PY' from t1

select * from V

ID        AMT TRX
-- ---------- ---
 1        100  PR
 2        100  PR
 3         50  PY
 4        100  PR
 5        200  PY
 6         50  PY
```

ID 列能唯一地标示每一笔交易。AMT 列代表每一笔交易涉及的金额（要么是还款，要么是购物）。TRX 列定义交易的类型：还款是 PY，购物是 PR。如果 TRX 的值是 PY，你希望能从累计值里减去当前的 AMT 值。如果 TRX 的值是 PR，你希望累计值加上当前的 AMT 值。最终，你想得到如下所示的结果集。

```
TRX_TYPE        AMT    BALANCE
-------- ---------- ----------
PURCHASE        100        100
```

```
PURCHASE        100        200
PAYMENT          50        150
PURCHASE        100        250
PAYMENT         200         50
PAYMENT          50          0
```

2. 解决方案

DB2 和 Oracle

使用窗口函数 SUM OVER 进行累计求和，并使用 CASE 表达式来决定交易的类型。

```
 1  select case when trx = 'PY'
 2              then 'PAYMENT'
 3              else 'PURCHASE'
 4          end trx_type,
 5          amt,
 6          sum(
 7           case when trx = 'PY'
 8              then -amt else amt
 9           end
10         ) over (order by id,amt) as balance
11    from V
```

MySQL、PostgreSQL 和 SQL Server

使用标量子查询进行累计求和，并使用 CASE 表达式来决定交易的类型。

```
 1  select case when v1.trx = 'PY'
 2              then 'PAYMENT'
 3              else 'PURCHASE'
 4          end as trx_type,
 5          v1.amt,
 6          (select sum(
 7                   case when v2.trx = 'PY'
 8                        then -v2.amt else v2.amt
 9                   end
10                 )
11            from V v2
12           where v2.id <= v1.id) as balance
13   from V v1
```

3. 讨论

CASE 表达式用于决定是把当前的 AMT 值加到累计值中，还是从累计值中减去当前的 AMT 值。如果交易类型是还款，AMT 值会被变为负数，因而相应的累计值会减少。CASE 表达式的结果如下所示。

```
select case when trx = 'PY'
            then 'PAYMENT'
            else 'PURCHASE'
       end trx_type,
       case when trx = 'PY'
            then -amt else amt
       end as amt
  from V
```

```
TRX_TYPE        AMT
-------- ---------
PURCHASE        100
PURCHASE        100
PAYMENT         -50
PURCHASE        100
PAYMENT        -200
PAYMENT         -50
```

依据交易类型的评估结果，AMT 值会被加入累计值，或者从累计值中减去。关于如何使用窗口函数 SUM OVER 或者标量子查询进行累计求和，请参考 7.6 节。

第 8 章

日期运算

本章介绍简单的日期运算技巧，其中的实例涵盖了一些常见任务，例如在给定日期的基础上添加若干天并算出新日期，找出两个日期之间的工作日个数，以及计算两个日期之间相差的天数。

熟练运用关系数据库管理系统的内置函数处理日期，有助于大幅度提高工作效率。在本章的实例里，我会尽量充分利用各种关系数据库管理系统的内置函数。除此之外，所有实例的日期都统一使用"日－月－年"格式。我之所以这么做，是想为那些平时工作中专注于一种关系数据库管理系统、同时又希望了解其他关系数据库管理系统的读者提供便利。统一使用标准格式使得大家不必花时间关心各个关系数据库管理系统默认的日期格式，这样就能把精力放在不同的方法和每种关系数据库管理系统提供的内置函数上。



本章主要介绍基本的日期运算。下一章的实例将介绍更多的高级技巧。本章的实例只会用到基本的日期数据类型。如果你要使用更复杂的日期数据类型，应在本章解决方案的基础上做出适当调整。

8.1 年月日加减法

1. 问题

你需要在给定日期的基础上加上或减去若干天、月或年。以员工 CLARK 的 `HIREDATE` 为例，你希望计算出 6 个不同的日期：CLARK 入职前后 5 天的日期，CLARK 入职前后 5 个月的日期，以及 CLARK 入职前后 5 年的日期。CLARK 的 `HIREDATE` 是"`09-JUN-1981`"（1981 年 6 月 9 日），你想得到如下所示的结果集。

```
HD_MINUS_5D HD_PLUS_5D  HD_MINUS_5M HD_PLUS_5M  HD_MINUS_5Y HD_PLUS_5Y
----------- ----------- ----------- ----------- ----------- -----------
04-JUN-1981 14-JUN-1981 09-JAN-1981 09-NOV-1981 09-JUN-1976 09-JUN-1986
```

```
12-NOV-1981 22-NOV-1981 17-JUN-1981 17-APR-1982 17-NOV-1976 17-NOV-1986
18-JAN-1982 28-JAN-1982 23-AUG-1981 23-JUN-1982 23-JAN-1977 23-JAN-1987
```

2. 解决方案

DB2

支持针对日期的加法和减法运算，但不论加上一个数还是减去一个数，后面都要指定对应的时间单位。

```
1 select hiredate -5 day   as hd_minus_5D,
2        hiredate +5 day   as hd_plus_5D,
3        hiredate -5 month as hd_minus_5M,
4        hiredate +5 month as hd_plus_5M,
5        hiredate -5 year  as hd_minus_5Y,
6        hiredate +5 year  as hd_plus_5Y
7   from emp
8  where deptno = 10
```

Oracle

若要加上或减去若干天，使用加法或减法即可。若要加减若干个月或年，则需要使用 ADD_MONTHS 函数。

```
1 select hiredate-5                 as hd_minus_5D,
2        hiredate+5                 as hd_plus_5D,
3        add_months(hiredate,-5)    as hd_minus_5M,
4        add_months(hiredate,5)     as hd_plus_5M,
5        add_months(hiredate,-5*12) as hd_minus_5Y,
6        add_months(hiredate,5*12)  as hd_plus_5Y
7   from emp
8  where deptno = 10
```

PostgreSQL

使用加减法，并使用 INTERVAL 关键字指定要加上或者减去的时间单位。在指定 INTERVAL 值的时候，必须使用英文单引号。

```
1 select hiredate - interval '5 day'   as hd_minus_5D,
2        hiredate + interval '5 day'   as hd_plus_5D,
3        hiredate - interval '5 month' as hd_minus_5M,
4        hiredate + interval '5 month' as hd_plus_5M,
5        hiredate - interval '5 year'  as hd_minus_5Y,
6        hiredate + interval '5 year'  as hd_plus_5Y
7   from emp
8  where deptno=10
```

MySQL

使用加减法，并使用 INTERVAL 关键字指定要加上或者减去的时间单位。不同于上述 PostgreSQL 的解决方案，指定 INTERVAL 值不必使用英文单引号。

```
1 select hiredate - interval 5 day   as hd_minus_5D,
2        hiredate + interval 5 day   as hd_plus_5D,
3        hiredate - interval 5 month as hd_minus_5M,
4        hiredate + interval 5 month as hd_plus_5M,
5        hiredate - interval 5 year  as hd_minus_5Y,
6        hiredate + interval 5 year  as hd_plus_5Y
```

```
7   from emp
8  where deptno=10
```

除此之外，还可以使用 DATE_ADD 函数，如下所示。

```
1 select date_add(hiredate,interval -5 day)   as hd_minus_5D,
2        date_add(hiredate,interval  5 day)   as hd_plus_5D,
3        date_add(hiredate,interval -5 month) as hd_minus_5M,
4        date_add(hiredate,interval  5 month) as hd_plus_5M,
5        date_add(hiredate,interval -5 year)  as hd_minus_5Y,
6        date_add(hiredate,interval  5 year)  as hd_plus_5DY
7   from emp
8  where deptno=10
```

SQL Server

使用 DATEADD 函数在给定日期值的基础上加上或者减去若个时间单位。

```
1 select dateadd(day,-5,hiredate)   as hd_minus_5D,
2        dateadd(day,5,hiredate)    as hd_plus_5D,
3        dateadd(month,-5,hiredate) as hd_minus_5M,
4        dateadd(month,5,hiredate)  as hd_plus_5M,
5        dateadd(year,-5,hiredate)  as hd_minus_5Y,
6        dateadd(year,5,hiredate)   as hd_plus_5Y
7   from emp
8  where deptno = 10
```

3. 讨论

Oracle 的解决方案利用了一个技巧：执行日期运算时，整数值代表天数。然而，这仅适用于 DATE 类型的运算。Oracle 9*i* 数据库引入了 TIMESTAMP 类型。对于这种新的日期数据类型，需要使用 PostgreSQL 解决方案中的 INTERVAL。如果要把 TIMESTAMP 类型传递给诸如 ADD_MONTHS 这样的旧式日期函数的话，还有一件事要注意：TIMESTAMP 值里面可能包含了精确到秒的数据，旧式的日期函数会忽略它们。

SQL 的 ISO 标准语法里规定了 INTERVAL 关键字以及紧随其后的字符串常量。该标准要求 INTERVAL 值必须位于英文单引号内。PostgreSQL（和 Oracle 9*i* 数据库及其后续版本）遵循了该标准。MySQL 则不支持英文单引号，略微偏离了标准。

8.2　计算两个日期之间的天数

1. 问题

找出两个日期之间相差多少天。例如，希望知道员工 ALLEN 和 WARD 的 HIREDATE 相差多少天。

2. 解决方案

DB2

使用内嵌视图找出 WARD 和 ALLEN 的 HIREDATE。然后，使用 DAYS 函数从一个 HIREDATE 里减去另一个。

```
1  select days(ward_hd) - days(allen_hd)
2    from (
```

```
 3  select hiredate as ward_hd
 4    from emp
 5   where ename = 'WARD'
 6         ) x,
 7         (
 8  select hiredate as allen_hd
 9    from emp
10   where ename = 'ALLEN'
11         ) y
```

Oracle 和 PostgreSQL

使用内嵌视图找出 WARD 和 ALLEN 的 HIREDATE，然后相减。

```
 1  select ward_hd - allen_hd
 2    from (
 3  select hiredate as ward_hd
 4    from emp
 5   where ename = 'WARD'
 6         ) x,
 7         (
 8  select hiredate as allen_hd
 9    from emp
10   where ename = 'ALLEN'
11         ) y
```

MySQL 和 SQL Server

使用 DATEDIFF 函数找出两个日期之间相差多少天。MySQL 的 DATEDIFF 函数只需要两个参数（两个将要相减的日期值），并且相对较早的日期值应该作为第一个参数以避免出现负数（SQL Server 正好相反）。SQL Server 的 DATEDIFF 函数可以返回指定的时间单位（本例中我们希望以天为单位）。下面给出了 SQL Server 的解决方案。

```
 1  select datediff(day,allen_hd,ward_hd)
 2    from (
 3  select hiredate as ward_hd
 4    from emp
 5   where ename = 'WARD'
 6         ) x,
 7         (
 8  select hiredate as allen_hd
 9    from emp
10   where ename = 'ALLEN'
11         ) y
```

对于 MySQL 而言，只需去掉 DATEDIFF 函数的第一个参数，并翻转 ALLEN_HD 和 WARD_HD 的顺序即可。

3. 讨论

上述全部解决方案中，内嵌视图 X 和 Y 被用于分别获取 WARD 和 ALLEN 的 HIREDATE。例如：

```
select ward_hd, allen_hd
  from (
select hiredate as ward_hd
  from emp
 where ename = 'WARD'
```

```
       ) y,
       (
select hiredate as allen_hd
  from emp
 where ename = 'ALLEN'
       ) x

WARD_HD     ALLEN_HD
----------- ---------
22-FEB-1981 20-FEB-1981
```

因为 X 和 Y 之间没有任何连接条件，这里会产生笛卡儿积。然后，由于本例中的 X 和 Y 都只有一条数据，因而即使没有连接条件也不会有问题，结果集最终只会有一行（很明显，因为 1 × 1=1）。为了计算两个日期之间相差多少天，只要使用适当的方法用一个日期减去另一个日期即可。

8.3 计算两个日期之间的工作日天数

1. 问题

给定两个日期，你想知道两者之间有多少个工作日，并且两个日期自身也要计算进去。例如，如果 1 月 10 日是星期一，1 月 11 日是星期二，则两者之间的工作日个数是 2，这是因为两个日期都是工作日。对于本实例而言，“工作日”定义为除了星期六和星期日以外的日子。

2. 解决方案

下面的解决方案以找出 BLAKE 和 JONES 的 HIREDATE 之间有多少个工作日为例。为了计算两个日期之间的工作日天数，我们可以使用数据透视表，把两个日期之间的每一天（包括开始和结束日期）都作为单独的一行返回。这样的话，统计工作日天数就变成了计算有多少个既不是星期六又不是星期日的日期。



如果想要把假日也排除在外，就需要创建一个 HOLIDAYS 表。然后在本解决方案基础上，使用 NOT IN 排除掉 HOLIDAYS 表里列出的日期。

DB2

使用数据透视表 T500 生成所需要的行数（两个日期之间的天数）的结果集。然后统计非周末日期的个数。使用 DAYNAME 函数能够知道一个日期是星期几。例如：

```
1  select sum(case when dayname(jones_hd+t500.id day -1 day)
2                    in ( 'Saturday','Sunday' )
3                   then 0 else 1
4              end) as days
5    from (
6  select max(case when ename = 'BLAKE'
7                   then hiredate
8              end) as blake_hd,
9         max(case when ename = 'JONES'
10                  then hiredate
```

```
11              end) as jones_hd
12    from emp
13   where ename in ( 'BLAKE','JONES' )
14          ) x,
15          t500
16   where t500.id <= blake_hd-jones_hd+1
```

MySQL

使用数据透视表 T500 生成所需要的行数（两个日期之间的天数）的结果集，然后统计非周末日期的个数。使用 DATE_ADD 函数为每一个日期加上若干天。使用 DATE_FORMAT 函数能够知道一个日期是星期几。

```
 1 select sum(case when date_format(
 2                         date_add(jones_hd,
 3                                  interval t500.id-1 DAY),'%a')
 4                   in ( 'Sat','Sun' )
 5                 then 0 else 1
 6            end) as days
 7   from (
 8 select max(case when ename = 'BLAKE'
 9                 then hiredate
10            end) as blake_hd,
11         max(case when ename = 'JONES'
12                  then hiredate
13             end) as jones_hd
14   from emp
15  where ename in ( 'BLAKE','JONES' )
16        ) x,
17        t500
18  where t500.id <= datediff(blake_hd,jones_hd)+1
```

Oracle

使用数据透视表 T500 生成所需要的行数（两个日期之间的天数）的结果集，然后统计非周末日期的个数。使用 TO_CHAR 函数能够知道一个日期是星期几。

```
 1 select sum(case when to_char(jones_hd+t500.id-1,'DY')
 2                   in ( 'SAT','SUN' )
 3                 then 0 else 1
 4            end) as days
 5   from (
 6 select max(case when ename = 'BLAKE'
 7                 then hiredate
 8            end) as blake_hd,
 9        max(case when ename = 'JONES'
10                 then hiredate
11            end) as jones_hd
12   from emp
13  where ename in ( 'BLAKE','JONES' )
14        ) x,
15        t500
16  where t500.id <= blake_hd-jones_hd+1
```

PostgreSQL

使用数据透视表 T500 生成所需要的行数（两个日期之间的天数）的结果集，然后统计非周

末日期的个数。使用 TO_CHAR 函数能够知道一个日期是星期几。

```
 1  select sum(case when trim(to_char(jones_hd+t500.id-1,'DAY'))
 2                    in ( 'SATURDAY','SUNDAY' )
 3                  then 0 else 1
 4             end) as days
 5    from (
 6  select max(case when ename = 'BLAKE'
 7                  then hiredate
 8             end) as blake_hd,
 9         max(case when ename = 'JONES'
10                  then hiredate
11             end) as jones_hd
12    from emp
13   where ename in ( 'BLAKE','JONES' )
14         ) x,
15         t500
16   where t500.id <= blake_hd-jones_hd+1
```

SQL Server

使用数据透视表 T500 生成所需要的行数（两个日期之间的天数）的结果集，然后统计非周末日期的个数。使用 DATENAME 函数能够知道一个日期是星期几。

```
 1  select sum(case when datename(dw,jones_hd+t500.id-1)
 2                    in ( 'SATURDAY','SUNDAY' )
 3                   then 0 else 1
 4             end) as days
 5    from (
 6  select max(case when ename = 'BLAKE'
 7                  then hiredate
 8             end) as blake_hd,
 9         max(case when ename = 'JONES'
10                  then hiredate
11             end) as jones_hd
12    from emp
13   where ename in ( 'BLAKE','JONES' )
14         ) x,
15         t500
16   where t500.id <= datediff(day,jones_hd-blake_hd)+1
```

3. 讨论

尽管不同数据库要调用不同的内置函数来确认一个日期是星期几，但上述解决方案的思路都是相同的。具体做法分为两步：

(1) 计算出开始日期和结束日期之间相隔多少天（包含开始日期和结束日期）；
(2) 排除掉周末，统计有多少个工作日（实际是在计算有多少条记录）。

内嵌视图 X 负责完成第一步工作。仔细查看内嵌视图 X 的话，我们会注意到它使用了聚合函数 MAX，其目的在于排除掉 Null。如果不了解 MAX 的用处，下面的查询结果输出能帮我们加深理解。下面的输出内容展示了没有 MAX 函数的内嵌视图 X 的查询结果。

```
select case when ename = 'BLAKE'
            then hiredate
```

```
       end as blake_hd,
       case when ename = 'JONES'
            then hiredate
       end as jones_hd
  from emp
 where ename in ( 'BLAKE','JONES' )

BLAKE_HD    JONES_HD
----------- -----------
            02-APR-1981
01-MAY-1981
```

如果不调用 MAX 函数，会有两行查询结果。有了 MAX 函数，查询结果就是一行，Null 会被过滤掉。

```
select max(case when ename = 'BLAKE'
            then hiredate
       end) as blake_hd,
       max(case when ename = 'JONES'
            then hiredate
       end) as jones_hd
  from emp
 where ename in ( 'BLAKE','JONES' )

BLAKE_HD    JONES_HD
----------- -----------
01-MAY-1981 02-APR-1981
```

两个日期之间相差 30 天（包含开始日期和结束日期）。现在，我们在同一行中得到了两个日期，下一步就是为这 30 天里的每一天单独生成一行记录。为了得到 30 天（行）的记录，这里用到了 T500 表。T500 表的 ID 列每一个值都等于前面一行的值加上 1，在两个日期中较早的那个（JONES_HD）的基础上依次加上 T500 表对应的值，这样就生成了 JONES_HD 和 BLAKE_HD 之间的连续日期序列。结果如下所示（使用 Oracle 语法）。

```
select x.*, t500.*, jones_hd+t500.id-1
  from (
select max(case when ename = 'BLAKE'
                then hiredate
           end) as blake_hd,
       max(case when ename = 'JONES'
                then hiredate
           end) as jones_hd
  from emp
 where ename in ( 'BLAKE','JONES' )
       ) x,
       t500
 where t500.id <= blake_hd-jones_hd+1

BLAKE_HD    JONES_HD            ID JONES_HD+T5
----------- ----------- ---------- -----------
01-MAY-1981 02-APR-1981          1 02-APR-1981
01-MAY-1981 02-APR-1981          2 03-APR-1981
01-MAY-1981 02-APR-1981          3 04-APR-1981
```

```
01-MAY-1981 02-APR-1981          4 05-APR-1981
01-MAY-1981 02-APR-1981          5 06-APR-1981
01-MAY-1981 02-APR-1981          6 07-APR-1981
01-MAY-1981 02-APR-1981          7 08-APR-1981
01-MAY-1981 02-APR-1981          8 09-APR-1981
01-MAY-1981 02-APR-1981          9 10-APR-1981
01-MAY-1981 02-APR-1981         10 11-APR-1981
01-MAY-1981 02-APR-1981         11 12-APR-1981
01-MAY-1981 02-APR-1981         12 13-APR-1981
01-MAY-1981 02-APR-1981         13 14-APR-1981
01-MAY-1981 02-APR-1981         14 15-APR-1981
01-MAY-1981 02-APR-1981         15 16-APR-1981
01-MAY-1981 02-APR-1981         16 17-APR-1981
01-MAY-1981 02-APR-1981         17 18-APR-1981
01-MAY-1981 02-APR-1981         18 19-APR-1981
01-MAY-1981 02-APR-1981         19 20-APR-1981
01-MAY-1981 02-APR-1981         20 21-APR-1981
01-MAY-1981 02-APR-1981         21 22-APR-1981
01-MAY-1981 02-APR-1981         22 23-APR-1981
01-MAY-1981 02-APR-1981         23 24-APR-1981
01-MAY-1981 02-APR-1981         24 25-APR-1981
01-MAY-1981 02-APR-1981         25 26-APR-1981
01-MAY-1981 02-APR-1981         26 27-APR-1981
01-MAY-1981 02-APR-1981         27 28-APR-1981
01-MAY-1981 02-APR-1981         28 29-APR-1981
01-MAY-1981 02-APR-1981         29 30-APR-1981
01-MAY-1981 02-APR-1981         30 01-MAY-1981
```

如果仔细查看 WHERE 子句的话，我们会注意到 BLAKE_HD 和 JONES_HD 相减后又加上了 1，这是为了生成所需的 30 行记录（否则就是 29 行）。我们也要注意到外层查询的 SELECT 列表里 T500.ID 减去了 1，这是因为 ID 列的起始值是 1，如果在 JONES_HD 基础上加上 1 就等同于从最终结果里排除掉了 JONES_HD。

一旦生成了所需数目的行记录，接着使用 CASE 表达式来标记每一个日期是工作日或者周末（若是工作日返回 1，周末则返回 0）。最后使用聚合函数 SUM 来合计 1 的个数，并得到最终答案。

8.4　计算两个日期之间相差的月份和年份

1. 问题

找出两个日期之间相差多少个月或者多少年。例如，希望知道第一个和最后一个员工的入职开始日期之间相差多少个月，同时也希望把这个差值换算成年。

2. 解决方案

一年有 12 个月，我们可以算出两个日期之间相差几个月，然后除以 12 得到相应的年份。在上述做法的基础上，可能有必要对年份的计算结果做向上或者向下的舍入。例如，EMP 表里最早的 HIREDATE 是 17-DEC-1980，最新的则是 12-JAN-1983。如果纯粹做数学运算，二者相差三年（1983 减去 1980），然而它们的实际差值只有大约 25 个月（2 年多一点）。我们应该根据需要调整做法，本节的解决方案给出的答案将会是 25 个月和大约 2 年。

DB2 和 MySQL

使用函数 YEAR 和 MONTH 计算出给定日期的含有 4 位数字的年份和含有 2 位数字的月份。

```
1  select mnth, mnth/12
2    from (
3  select (year(max_hd) - year(min_hd))*12 +
4         (month(max_hd) - month(min_hd)) as mnth
5    from (
6  select min(hiredate) as min_hd, max(hiredate) as max_hd
7    from emp
8         ) x
9         ) y
```

Oracle

使用 MONTHS_BETWEEN 函数找出两个日期之间相差多少个月（再除以 12 就可以得到相差多少年）。

```
1  select months_between(max_hd,min_hd),
2         months_between(max_hd,min_hd)/12
3    from (
4  select min(hiredate) min_hd, max(hiredate) max_hd
5    from emp
6         ) x
```

PostgreSQL

使用 EXTRACT 函数计算出给定日期的含有 4 位数字的年份和含有 2 位数字的月份。

```
1  select mnth, mnth/12
2    from (
3  select ( extract(year from max_hd) -
4           extract(year from min_hd) ) * 12
5         +
6         ( extract(month from max_hd) -
7           extract(month from min_hd) ) as mnth
8    from (
9  select min(hiredate) as min_hd, max(hiredate) as max_hd
10   from emp
11        ) x
12        ) y
```

SQL Server

使用 DATEDIFF 函数找出两个日期之间相差多少个月（再除以 12 就可以得到相差多少年）。

```
1  select datediff(month,min_hd,max_hd),
2         datediff(month,min_hd,max_hd)/12
3    from (
4  select min(hiredate) min_hd, max(hiredate) max_hd
5    from emp
6         ) x
```

3. 讨论

DB2、MySQL 和 PostgreSQL

一旦提取出 MIN_HD 和 MAX_HD 的年份和月份，计算 MIN_HD 和 MAX_HD 之间相差的月份和年份的方法对于所有数据库都将是相同的。接下来的讨论将涵盖这 3 种数据库。内嵌视图 X 将

找出 EMP 表里最早的和最新的 HIREDATE，如下所示。

```
select min(hiredate) as min_hd,
       max(hiredate) as max_hd
  from emp

MIN_HD      MAX_HD
----------- -----------
17-DEC-1980 12-JAN-1983
```

为了找出 MAX_HD 和 MIN_HD 之间相差的月份，先用它们相差的年份乘以 12，然后再加上月份的差值。如果你还是不太理解这样做的原因，不妨分别打印出每个日期的年和月两个部分。下面显示了它们的值。

```
select year(max_hd)  as max_yr,   year(min_hd) as min_yr,
       month(max_hd) as max_mon, month(min_hd) as min_mon
  from (
select min(hiredate) as min_hd, max(hiredate) as max_hd
  from emp
       ) x

MAX_YR     MIN_YR    MAX_MON    MIN_MON
------ ---------- ---------- ----------
  1983       1980          1         12
```

根据上述结果，MAX_HD 和 MIN_HD 之间相差的月份就是 (1983−1980) × 12 ＋ (1−12)。为了找出两个日期之间相差的年份，就把前面计算出来的月份差值除以 12。再次强调，我们可能需要根据实际状况对年份的计算结果做出适当地舍入处理。

Oracle 和 SQL Server

内嵌视图 X 检索 EMP 表里最早的和最新的 HIREDATE，如下所示。

```
select min(hiredate) as min_hd, max(hiredate) as max_hd
  from emp

MIN_HD      MAX_HD
----------- -----------
17-DEC-1980 12-JAN-1983
```

Oracle 和 SQL Server 提供了函数（分别是 MONTHS_BETWEEN 和 DATEDIFF）用于计算两个给定日期之间相差的月份。为了计算相差的年份，把月份差值除以 12 即可。

8.5　计算两个日期之间相差的秒数、分钟数和小时数

1. 问题

算出两个日期之间相差多少秒。例如，希望知道 ALLEN 和 WARD 的 HIREDATE 之间相差多少秒、多少分钟以及多少小时。

2. 解决方案

如果我们能算出来两个日期之间相差多少天，那么也就能知道它们相差多少秒、多少分钟和多少小时，只需对不同的时间单位做出相应的换算即可。

DB2

使用 DAYS 函数计算 ALLEN 和 WARD 的 HIREDATE 之间相差多少天，然后进行时间单位换算。

```
1 select dy*24 hr, dy*24*60 min, dy*24*60*60 sec
2   from (
3 select ( days(max(case when ename = 'WARD'
4                   then hiredate
5              end)) -
6          days(max(case when ename = 'ALLEN'
7                   then hiredate
8              end))
9        ) as dy
10   from emp
11        ) x
```

MySQL 和 SQL Server

使用 DATEDIFF 函数计算 ALLEN 和 WARD 的 HIREDATE 之间相差多少天，然后进行时间单位换算。

```
1 select datediff(day,allen_hd,ward_hd)*24 hr,
2        datediff(day,allen_hd,ward_hd)*24*60 min,
3        datediff(day,allen_hd,ward_hd)*24*60*60 sec
4   from (
5 select max(case when ename = 'WARD'
6                  then hiredate
7            end) as ward_hd,
8        max(case when ename = 'ALLEN'
9                  then hiredate
10           end) as allen_hd
11   from emp
12        ) x
```

Oracle 和 PostgreSQL

使用减法计算 ALLEN 和 WARD 的 HIREDATE 之间相差多少天，然后进行时间单位换算。

```
1 select dy*24 as hr, dy*24*60 as min, dy*24*60*60 as sec
2   from (
3 select (max(case when ename = 'WARD'
4                 then hiredate
5             end) -
6         max(case when ename = 'ALLEN'
7                  then hiredate
8             end)) as dy
9    from emp
10        ) x
```

3. 讨论

在上述所有的解决方案中，内嵌视图 X 都被用来获取 WARD 和 ALLEN 的 HIREDATE，如下所示。

```
select max(case when ename = 'WARD'
                 then hiredate
           end) as ward_hd,
       max(case when ename = 'ALLEN'
                then hiredate
           end) as allen_hd
  from emp

WARD_HD     ALLEN_HD
----------- -----------
22-FEB-1981 20-FEB-1981
```

WARD_HD 和 ALLEN_HD 之间相差的天数分别乘以 24（一天的小时数），1440（一天的分钟数）和 86400（一天的秒数），就得到最终结果了。

8.6 统计一年中有多少个星期一

1. 问题

你想知道一年中有多少个星期一、星期二、星期三等。

2. 解决方案

为了统计出一年里每个“星期 *x*”出现的次数，我们必须：

(1) 生成一年里所有可能的日期值；
(2) 格式化上述日期值，并找出它们分别是星期几；
(3) 统计每个“星期 *x*”出现的次数。

DB2

使用 WITH 递归查询，这样就不需要对一个至少含有 366 行记录的表做 SELECT 查询了。使用 DAYNAME 函数获知每一个日期是星期几，然后统计每个“星期 *x*”出现的次数。

```
1  with x (start_date,end_date)
2  as (
3  select start_date,
4         start_date + 1 year end_date
5    from (
6  select (current_date -
7          dayofyear(current_date) day)
8          +1 day as start_date
9    from t1
10        )tmp
11  union all
12 select start_date + 1 day, end_date
13   from x
14  where start_date + 1 day < end_date
15 )
16 select dayname(start_date),count(*)
17   from x
18  group by dayname(start_date)
```

MySQL

针对 T500 表做 SELECT 查询以产生出足够行数的结果集，每一行记录代表一年中的一个日期。使用 DATE_FORMAT 函数获知每一个日期是星期几，然后统计每个“星期 *x*”出现的次数。

```
1  select date_format(
2            date_add(
3                cast(
4              concat(year(current_date),'-01-01')
5                     as date),
6                     interval t500.id-1 day),
7                     '%W') day,
8         count(*)
9    from t500
10  where t500.id <= datediff(
11                        cast(
12                      concat(year(current_date)+1,'-01-01')
13                             as date),
14                        cast(
15                      concat(year(current_date),'-01-01')
16                             as date))
17  group by date_format(
18               date_add(
19                   cast(
20                 concat(year(current_date),'-01-01')
21                        as date),
22                        interval t500.id-1 day),
23                        '%W')
```

Oracle

对于 Oracle 9*i* 及其后续版本，可以使用 CONNECT BY 递归查询返回一年中的每一天。如果是 Oracle 8*i* 或更早版本，则需要针对 T500 表做 SELECT 查询以产生出足够行数的结果集，每一行记录代表一年中的一个日期。不论哪种方法，都需要使用 TO_CHAR 函数以获知每一个日期是星期几，然后统计每个“星期 *x*”出现的次数。

首先来看一下 CONNECT BY 解决方案。

```
1 with x as (
2 select level lvl
3   from dual
4  connect by level <= (
5    add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
6  )
7 )
8 select to_char(trunc(sysdate,'y')+lvl-1,'DAY'), count(*)
9   from x
10  group by to_char(trunc(sysdate,'y')+lvl-1,'DAY')
```

其次是针对 Oracle 早期版本的解决方案。

```
1 select to_char(trunc(sysdate,'y')+rownum-1,'DAY'),
2        count(*)
3   from t500
```

```
4  where rownum <= (add_months(trunc(sysdate,'y'),12)
5                    - trunc(sysdate,'y'))
6  group by to_char(trunc(sysdate,'y')+rownum-1,'DAY')
```

PostgreSQL

使用内置函数 GENERATE_SERIES 为一年中的每一天生成一行记录。然后使用 TO_CHAR 函数获知每一个日期是星期几。最后，统计每个“星期 *x*”出现的次数。例如：

```
1  select to_char(
2            cast(
3      date_trunc('year',current_date)
4                  as date) + gs.id-1,'DAY'),
5         count(*)
6    from generate_series(1,366) gs(id)
7   where gs.id <= (cast
8                     ( date_trunc('year',current_date) +
9                          interval '12 month' as date) -
10 cast(date_trunc('year',current_date)
11                        as date))
12  group by to_char(
13               cast(
14         date_trunc('year',current_date)
15            as date) + gs.id-1,'DAY')
```

SQL Server

使用 WITH 递归查询，这样就不需要对一个至少含有 366 行记录的表做 SELECT 查询了。如果是早期不支持 WITH 子句的 SQL Server 版本，参考 Oracle 解决方案里使用数据透视表的做法。使用 DATENAME 函数获知一个日期是星期几，然后统计每个“星期 *x*”出现的次数。例如：

```
1  with x (start_date,end_date)
2  as (
3  select start_date,
4         dateadd(year,1,start_date) end_date
5    from (
6  select cast(
7         cast(year(getdate()) as varchar) + '-01-01'
8              as datetime) start_date
9    from t1
10        ) tmp
11 union all
12 select dateadd(day,1,start_date), end_date
13   from x
14  where dateadd(day,1,start_date) < end_date
15 )
16 select datename(dw,start_date),count(*)
17   from x
18  group by datename(dw,start_date)
19 OPTION (MAXRECURSION 366)
```

3. 讨论

DB2

WITH 递归查询视图 X 里的内嵌视图 TMP 返回当前年份第一天的日期，如下所示。

```
select (current_date -
        dayofyear(current_date) day)
        +1 day as start_date
  from t1

START_DATE
-----------
01-JAN-2005
```

然后在 START_DATE 基础上加上一年，这样我们就知道了这一年的第一天和最后一天的日期。我们需要知道这两个日期，因为要生成这一年的所有日期。START_DATE 和 END_DATE 如下所示。

```
select start_date,
       start_date + 1 year end_date
  from (
select (current_date -
        dayofyear(current_date) day)
        +1 day as start_date
  from t1
       ) tmp

START_DATE  END_DATE
----------- -----------
01-JAN-2005 01-JAN-2006
```

下一步是为 START_DATE 加上一天，如此循环往复，直到它等于 END_DATE。下面展示了 WITH 递归查询视图 X 返回的结果集的一部分。

```
with x (start_date,end_date)
as (
select start_date,
       start_date + 1 year end_date
  from (
select (current_date -
        dayofyear(current_date) day)
        +1 day as start_date
  from t1
       ) tmp
 union all
select start_date + 1 day, end_date
  from x
 where start_date + 1 day < end_date
)
select * from x

START_DATE  END_DATE
----------- -----------
01-JAN-2005 01-JAN-2006
02-JAN-2005 01-JAN-2006
03-JAN-2005 01-JAN-2006
...
29-JAN-2005 01-JAN-2006
30-JAN-2005 01-JAN-2006
```

```
31-JAN-2005 01-JAN-2006
...
01-DEC-2005 01-JAN-2006
02-DEC-2005 01-JAN-2006
03-DEC-2005 01-JAN-2006
...
29-DEC-2005 01-JAN-2006
30-DEC-2005 01-JAN-2006
31-DEC-2005 01-JAN-2006
```

最后，针对 WITH 递归查询视图 X 的返回结果调用 DAYNAME 函数，并统计每个“星期 *x*”出现的次数。最终结果如下所示。

```
with x (start_date,end_date)
as (
select start_date,
       start_date + 1 year end_date
  from (
select (current_date -
        dayofyear(current_date) day)
        +1 day as start_date
  from t1
       ) tmp
 union all
select start_date + 1 day, end_date
  from x
 where start_date + 1 day < end_date
)
select dayname(start_date),count(*)
  from x
 group by dayname(start_date)

START_DATE   COUNT(*)
---------  ----------
FRIDAY             52
MONDAY             52
SATURDAY           53
SUNDAY             52
THURSDAY           52
TUESDAY            52
WEDNESDAY          52
```

MySQL

本解决方案结合 T500 表执行 SELECT 查询，为一年中的每一个日期生成单独的一行数据。第 4 行的命令用于生成当前年份的第一天。它的做法是调用 CURRENT_DATE 函数得到年份，然后在后面附加上月份和天（遵守 MySQL 默认的日期格式）。结果如下所示。

```
select concat(year(current_date),'-01-01')
  from t1

START_DATE
-----------
01-JAN-2005
```

现在有了当前年份第一天的日期，调用 DATEADD 函数将其与 T500.ID 逐一相加以生成一年里的每一天。调用 DATE_FORMAT 函数能够返回每一个日期是星期几。为了以 T500 表为基础生成足够多的行，要先找出当前年份第一天和下一年度第一天之间的差值，并生成与之相等数目的行（应该是 365 行或者 366 行）。部分查询结果如下所示。

```
select date_format(
          date_add(
              cast(
            concat(year(current_date),'-01-01')
                   as date),
                   interval t500.id-1 day),
                   '%W') day
  from t500
 where t500.id <= datediff(
                      cast(
                    concat(year(current_date)+1,'-01-01')
                           as date),
                      cast(
                    concat(year(current_date),'-01-01')
                           as date))

DAY
-----------
01-JAN-2005
02-JAN-2005
03-JAN-2005
...
29-JAN-2005
30-JAN-2005
31-JAN-2005
...
01-DEC-2005
02-DEC-2005
03-DEC-2005
...
29-DEC-2005
30-DEC-2005
31-DEC-2005
```

现在有了当前年份每一天的日期了，接下来要调用 DAYNAME 函数得到每一个日期是星期几，并统计每个“星期 x”出现的次数。最终结果如下所示。

```
select date_format(
          date_add(
              cast(
            concat(year(current_date),'-01-01')
                   as date),
                   interval t500.id-1 day),
                   '%W') day,
       count(*)
  from t500
 where t500.id <= datediff(
                      cast(
                    concat(year(current_date)+1,'-01-01')
```

```
                        as date),
                   cast(
                 concat(year(current_date),'-01-01')
                        as date))
 group by date_format(
            date_add(
                cast(
              concat(year(current_date),'-01-01')
                   as date),
                   interval t500.id-1 day),
                   '%W')
```

```
DAY          COUNT(*)
--------- ----------
FRIDAY            52
MONDAY            52
SATURDAY          53
SUNDAY            52
THURSDAY          52
TUESDAY           52
WEDNESDAY         52
```

Oracle

有两种解决方案：结合 T500 表（数据透视表）执行 SELECT 查询，或者使用 CONNECT BY 和 WITH 递归查询，都能为当前年份的每一个日期生成单独的一行数据。使用 TRUNC 函数把当前的系统日期转换为当前年份的第一天。

对于 CONNECT BY 和 WITH 解决方案，我们利用临时的 LEVEL 列生成从 1 开始的数字序列。为了生成足够多行的数据，根据当前年份第一天和下一年度第一天之间的差值（应该是 365 天或者 366 天）筛选 ROWNUM 或者 LEVEL。下一步就是在当前年份第一天的基础上依次加上 ROWNUM 或者 LEVEL。部分查询结果如下所示。

```
/* Oracle 9i及后续版本 */
with x as (
select level lvl
  from dual
 connect by level <= (
   add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
 )
)
select trunc(sysdate,'y')+lvl-1
  from x
```

对于使用数据透视表的解决方案，我们可以使用任何行数超过 366 行的表或者视图。由于 Oracle 支持 ROWNUM，我们并不需要一个从 1 开始递增的表。在下面的例子中，我们利用数据透视表 T500 来返回当前年份的每一天。

```
/* Oracle 8i及更早版本 */
select trunc(sysdate,'y')+rownum-1 start_date
  from t500
 where rownum <= (add_months(trunc(sysdate,'y'),12)
                 - trunc(sysdate,'y'))
```

```
START_DATE
-----------
01-JAN-2005
02-JAN-2005
03-JAN-2005
...
29-JAN-2005
30-JAN-2005
31-JAN-2005
...
01-DEC-2005
02-DEC-2005
03-DEC-2005
...
29-DEC-2005
30-DEC-2005
31-DEC-2005
```

不论是哪种解决方案，最终我们必须调用 TO_CHAR 函数获取每一个日期分别是星期几，然后统计每个“星期 x”出现的次数。最终结果如下所示。

```
/* Oracle 9i及后续版本 */
with x as (
select level lvl
  from dual
 connect by level <= (
   add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
 )
)
select to_char(trunc(sysdate,'y')+lvl-1,'DAY'), count(*)
  from x
 group by to_char(trunc(sysdate,'y')+lvl-1,'DAY')

 /* Oracle 8i及更早版本 */
select to_char(trunc(sysdate,'y')+rownum-1,'DAY') start_date,
       count(*)
  from t500
 where rownum <= (add_months(trunc(sysdate,'y'),12)
                  - trunc(sysdate,'y'))
 group by to_char(trunc(sysdate,'y')+rownum-1,'DAY')

START_DATE   COUNT(*)
---------- ----------
FRIDAY             52
MONDAY             52
SATURDAY           53
SUNDAY             52
THURSDAY           52
TUESDAY            52
WEDNESDAY          52
```

PostgreSQL

首先调用 DATE_TRUNC 函数获取当前的年份（如下所示，由于针对 T1 做查询，因而只返回一行结果）。

```
select cast(
         date_trunc('year',current_date)
       as date) as start_date
  from t1

START_DATE
-----------
01-JAN-2005
```

然后，针对一个至少有 366 行的数据源（任何表达式均可）执行 SELECT 查询。本解决方案采用了 GENERATE_SERIES 函数作为数据源。当然，这里也可以用 T500 表。然后，在当前年份第一天的基础上逐次加上 1，直到把一年中的每一天都作为单独的一行返回（如下所示）。

```
select cast( date_trunc('year',current_date)
               as date) + gs.id-1 as start_date
  from generate_series (1,366) gs(id)
 where gs.id <= (cast
                  ( date_trunc('year',current_date) +
                       interval '12 month' as date) -
     cast(date_trunc('year',current_date)
                      as date))

START_DATE
-----------
01-JAN-2005
02-JAN-2005
03-JAN-2005
...
29-JAN-2005
30-JAN-2005
31-JAN-2005
...
01-DEC-2005
02-DEC-2005
03-DEC-2005
...
29-DEC-2005
30-DEC-2005
31-DEC-2005
```

最后，调用 TO_CHAR 函数获取每一个日期分别是星期几，然后统计每个“星期 x”出现的次数。最终的结果如下所示。

```
select to_char(
          cast(
    date_trunc('year',current_date)
                as date) + gs.id-1,'DAY') as start_dates,
       count(*)
  from generate_series(1,366) gs(id)
 where gs.id <= (cast
                  ( date_trunc('year',current_date) +
                       interval '12 month' as date) -
     cast(date_trunc('year',current_date)
```

```
                     as date))
 group by to_char(
             cast(
       date_trunc('year',current_date)
          as date) + gs.id-1,'DAY')

START_DATE   COUNT(*)
---------- ----------
FRIDAY             52
MONDAY             52
SATURDAY           53
SUNDAY             52
THURSDAY           52
TUESDAY            52
WEDNESDAY          52
```

SQL Server

WITH 递归查询视图 X 里的内嵌视图 TMP 返回当前年份的第一天，如下所示。

```
select cast(
       cast(year(getdate()) as varchar) + '-01-01'
            as datetime) start_date
  from t1

START_DATE
-----------
01-JAN-2005
```

得到当前年份的第一天之后，再为 START_DATE 加上一年，这样我们就知道了开始日期和结束日期。我们需要知道这两个日期，因为需要生成这一年中的每一天。START_DATE 和 END_DATE 如下所示。

```
select start_date,
        dateadd(year,1,start_date) end_date
  from (
select cast(
       cast(year(getdate()) as varchar) + '-01-01'
            as datetime) start_date
  from t1
       ) tmp

START_DATE  END_DATE
----------- -----------
01-JAN-2005 01-JAN-2006
```

下一步，为 START_DATE 加上一天，如此循环往复，直到它等于 END_DATE。如下展示了 WITH 递归查询视图 X 返回的结果集的一部分。

```
with x (start_date,end_date)
 as (
 select start_date,
        dateadd(year,1,start_date) end_date
   from (
 select cast(
```

```
       cast(year(getdate()) as varchar) + '-01-01'
            as datetime) start_date
  from t1
       ) tmp
 union all
 select dateadd(day,1,start_date), end_date
   from x
  where dateadd(day,1,start_date) < end_date
 )
 select * from x
 OPTION (MAXRECURSION 366)

START_DATE  END_DATE
----------- -----------
01-JAN-2005 01-JAN-2006
02-JAN-2005 01-JAN-2006
03-JAN-2005 01-JAN-2006
...
29-JAN-2005 01-JAN-2006
30-JAN-2005 01-JAN-2006
31-JAN-2005 01-JAN-2006
...
01-DEC-2005 01-JAN-2006
02-DEC-2005 01-JAN-2006
03-DEC-2005 01-JAN-2006
...
29-DEC-2005 01-JAN-2006
30-DEC-2005 01-JAN-2006
31-DEC-2005 01-JAN-2006
```

最后一步是针对 WITH 递归查询视图 X 的返回结果调用 DAYNAME 函数，并统计每个“星期 *x*”出现的次数。最终结果如下所示。

```
with x(start_date,end_date)
 as (
 select start_date,
        dateadd(year,1,start_date) end_date
   from (
 select cast(
        cast(year(getdate()) as varchar) + '-01-01'
             as datetime) start_date
   from t1
        ) tmp
  union all
 select dateadd(day,1,start_date), end_date
   from x
  where dateadd(day,1,start_date) < end_date
 )
 select datename(dw,start_date), count(*)
   from x
  group by datename(dw,start_date)
 OPTION (MAXRECURSION 366)

START_DATE   COUNT(*)
---------  ----------
```

```
FRIDAY              52
MONDAY              52
SATURDAY            53
SUNDAY              52
THURSDAY            52
TUESDAY             52
WEDNESDAY           52
```

8.7 计算当前记录和下一条记录之间的日期差

1. 问题

计算两个日期之间相差多少天（特别是当两者分别存储于不同的行的时候）。例如，对于 DEPTNO 等于 10 的部门的每一个员工，你希望计算出他们的入职日期之间相差多少天。

2. 解决方案

要解决本问题有一个诀窍是，在早于当前入职时间的所有记录里找出 HIREDATE 的最小值。如此一来，剩下的工作就是利用 8.2 节里提到的技巧来计算出两个日期之间相差多少天。

DB2

使用标量子查询找出相对于当前 HIREDATE 的下一个 HIREDATE。然后，调用 DAYS 函数计算两个日期之间相差多少天。

```
1 select x.*,
2        days(x.next_hd) - days(x.hiredate) diff
3   from (
4 select e.deptno, e.ename, e.hiredate,
5        (select min(d.hiredate) from emp d
6          where d.hiredate > e.hiredate) next_hd
7   from emp e
8  where e.deptno = 10
9        ) x
```

MySQL 和 SQL Server

使用标量子查询找出相对于当前 HIREDATE 的下一个 HIREDATE。然后，调用 DATEDIFF 函数计算两个日期之间相差多少天。下面的代码里以 SQL Server 版本的 DATEDIFF 函数为例。

```
1 select x.*,
2        datediff(day,x.hiredate,x.next_hd) diff
3   from (
4 select e.deptno, e.ename, e.hiredate,
5        (select min(d.hiredate) from emp d
6          where d.hiredate > e.hiredate) next_hd
7   from emp e
8  where e.deptno = 10
9        ) x
```

对于 MySQL 版本的 DATEDIFF 函数，我们需要省略第一个参数 day，并把剩下的两个参数的顺序颠倒过来。

```
2        datediff(x.next_hd, x.hiredate) diff
```

Oracle

对于 Oracle 8*i* 及后续版本，使用窗口函数 LEAD OVER 访问相对于当前 HIREDATE 的下一个 HIREDATE，然后执行减法运算。

```
1  select ename, hiredate, next_hd,
2         next_hd - hiredate diff
3    from (
4  select deptno, ename, hiredate,
5         lead(hiredate)over(order by hiredate) next_hd
6    from emp
7         )
8   where deptno=10
```

对于 Oracle 8*i* 及更早的版本，则需要采用下面的 PostgreSQL 解决方案。

PostgreSQL

使用标量子查询找出相对于当前 HIREDATE 的下一个 HIREDATE。然后，直接利用减法运算得出两者相差多少天。

```
1 select x.*,
2        x.next_hd - x.hiredate as diff
3   from (
4 select e.deptno, e.ename, e.hiredate,
5        (select min(d.hiredate) from emp d
6          where d.hiredate > e.hiredate) as next_hd
7   from emp e
8  where e.deptno = 10
9        ) x
```

3. 讨论

DB2、MySQL、PostgreSQL 和 SQL Server

除了语法上的差别，所有这些解决方案的做法是相同的：使用标量子查询找出相对于当前 HIREDATE 的下一个 HIREDATE，然后使用本章 8.2 节里用过的技巧计算出两个日期之间相差多少天。

Oracle

窗口函数 LEAD OVER 非常有用，它可以访问“未来”的行（“未来”是相对于当前行而言的，由 ORDER BY 子句决定）。不必使用额外的连接查询就可以访问当前行前后的行数据，这种能力有助于我们写出更高效、可读性更好的代码。当使用窗口函数时，要记住它们是在 WHERE 子句之后才被评估执行的，这就是为什么本解决方案需要一个内嵌视图。如果我们把过滤 DEPTNO 的动作移到内嵌视图里面的话，结果就不一样了（就变成了只考虑 DEPTNO 等于 10 的员工的 HIREDATE）。关于 Oracle 的 LEAD 和 LAG 函数，需要特别指出的一点是它们对于重复项的处理。在前言部分我提到过，本书的实例都不包含“防御性代码”，因为有太多无法预见的状况都可能导致代码无法正常执行。也就是说，即使我们能预见到每一种可能出现的问题，但最终写出的 SQL 却可能已经冗繁到不具有可读性。因此，在大多数情况下，一个解决方案的意义在于它提供了一种技巧：我们能将其用于线上系统，但是又必须事先做好测试，并针对具体的数据做出必要的调整。对于本例而言，有一种情况稍后需要简单讨论一下，因为针对它的变通方案不是那么显而易见，对于不熟悉 Oracle 的

读者而言尤其如此。本例中 EMP 表里不存在重复的 HIREDATE，但是在一个表里出现重复的日期当然是可能的（并且非常可能）。考虑 DEPTNO 等于 10 的员工及其对应的 HIREDATE。

```
select ename, hiredate
  from emp
 where deptno=10
 order by 2

ENAME  HIREDATE
------ -----------
CLARK  09-JUN-1981
KING   17-NOV-1981
MILLER 23-JAN-1982
```

为了讨论的需要，我们插入 4 条重复数据，这样就有（包括 KING 在内的）5 个员工的 HIREDATE 都是 11 月 17 日了。

```
insert into emp (empno,ename,deptno,hiredate)
values (1,'ant',10,to_date('17-NOV-1981'))

insert into emp (empno,ename,deptno,hiredate)
values (2,'joe',10,to_date('17-NOV-1981'))

insert into emp (empno,ename,deptno,hiredate)
values (3,'jim',10,to_date('17-NOV-1981'))

insert into emp (empno,ename,deptno,hiredate)
values (4,'choi',10,to_date('17-NOV-1981'))

select ename, hiredate
  from emp
 where deptno=10
 order by 2

ENAME  HIREDATE
------ -----------
CLARK  09-JUN-1981
ant    17-NOV-1981
joe    17-NOV-1981
KING   17-NOV-1981
jim    17-NOV-1981
choi   17-NOV-1981
MILLER 23-JAN-1982
```

现在 DEPTNO 等于 10 的部门里就有不止一个人的 HIREDATE 是同一天了。如果仍然使用上述解决方案（但是要把 DEPTNO 过滤动作移到内嵌视图里面去，我们只关注 DEPTNO 等于 10 的员工及其 HIREDATE），返回的结果集就变成如下所示的输出内容。

```
select ename, hiredate, next_hd,
       next_hd - hiredate diff
  from (
select deptno, ename, hiredate,
       lead(hiredate)over(order by hiredate) next_hd
  from emp
```

```
 where deptno=10
        )

ENAME  HIREDATE    NEXT_HD           DIFF
------ ----------- ----------- ----------
CLARK  09-JUN-1981 17-NOV-1981        161
ant    17-NOV-1981 17-NOV-1981          0
joe    17-NOV-1981 17-NOV-1981          0
KING   17-NOV-1981 17-NOV-1981          0
jim    17-NOV-1981 17-NOV-1981          0
choi   17-NOV-1981 23-JAN-1982         67
MILLER 23-JAN-1982 (null)          (null)
```

看一下 HIREDATE 相同的 5 个员工，他们中有 4 个人的 DIFF 值是 0。这显然不正确。对于 HIREDATE 相同的员工而言，应该与下一个不同的 HIREDATE 相比较，例如 HIREDATE 是 11 月 17 日的员工要与 MILLER 的 HIREDATE 相比较。问题的根源在于 LEAD 函数仅仅按照 HIREDATE 排序，却不会自动去重。因此，如果把 ANT 的 HIREDATE 与 JOE 的相比较，相差的天数就是 0，如此一来 ANT 的 DIFF 值就变成 0 了。所幸 Oracle 已经为这种情况提供了一个简单的变通方案。当调用 LEAD 函数的时候，我们可以传递一个参数用于指定“未来行”的位置（例如，它位于下一行，或者隔了 10 行之后，等等）。因此，对于员工 ANT，需要跨过 5 行，而不是一行（我们希望跳过所有重复的 HIREDATE），去看一下 MILLER 的 HIREDATE。员工 JOE 距离 MILLER 有 4 行的距离，JIM 则是 3 行，KING 是 2 行，CHOI 幸运地只相隔一行。为了获取正确答案，只要把上述每一个员工所在行到 MILLER 的距离值作为参数传递给 LEAD 函数即可。该解决方案如下所示。

```
select ename, hiredate, next_hd,
       next_hd - hiredate diff
  from (
select deptno, ename, hiredate,
       lead(hiredate,cnt-rn+1)over(order by hiredate) next_hd
  from (
select deptno,ename,hiredate,
       count(*)over(partition by hiredate) cnt,
       row_number()over(partition by hiredate order by empno) rn
  from emp
 where deptno=10
       )
       )

ENAME  HIREDATE    NEXT_HD           DIFF
------ ----------- ----------- ----------
CLARK  09-JUN-1981 17-NOV-1981        161
ant    17-NOV-1981 23-JAN-1982         67
joe    17-NOV-1981 23-JAN-1982         67
jim    17-NOV-1981 23-JAN-1982         67
choi   17-NOV-1981 23-JAN-1982         67
KING   17-NOV-1981 23-JAN-1982         67
MILLER 23-JAN-1982 (null)          (null)
```

现在我们能得到正确的计算结果了。所有 HIREDATE 相同的员工都会和下一个不同的 HIREDATE 相比较，而不会匹配到一个相同的 HIREDATE 值。如果这个替代方案不是那么容易

理解，不妨把上述查询分解开来看。先从内嵌视图开始。

```
select deptno,ename,hiredate,
       count(*)over(partition by hiredate) cnt,
       row_number()over(partition by hiredate order by empno) rn
  from emp
 where deptno=10

DEPTNO ENAME  HIREDATE           CNT         RN
------ ------ ----------- ---------- ----------
    10 CLARK  09-JUN-1981          1          1
    10 ant    17-NOV-1981          5          1
    10 joe    17-NOV-1981          5          2
    10 jim    17-NOV-1981          5          3
    10 choi   17-NOV-1981          5          4
    10 KING   17-NOV-1981          5          5
    10 MILLER 23-JAN-1982          1          1
```

窗口函数 COUNT OVER 计算每一种 HIREDATE 值出现的次数，并为每一行记录返回该值。对于那个重复的 HIREDATE，CNT 的值都是 5。窗口函数 ROW_NUMBER OVER 按照 EMPNO 为每一个员工排名。排名按照 HIREDATE 分区，除非有重复的 HIREDATE 出现，否则每个员工对应的 RN 值都是 1。现在，所有重复数据都被分组计数并算出其在分组里的排名，而该排名值可以用于度量当前 HIREDATE 到下一个 HIREDATE（MILLER 的 HIREDATE）的距离。调用 LEAD 函数时，我们通过从 CNT 里减去 RN 值并加 1 来得出该距离值。

```
select deptno, ename, hiredate,
       cnt-rn+1 distance_to_miller,
       lead(hiredate,cnt-rn+1)over(order by hiredate) next_hd
  from (
select deptno,ename,hiredate,
       count(*)over(partition by hiredate) cnt,
       row_number()over(partition by hiredate order by empno) rn
  from emp
 where deptno=10
        )

DEPTNO ENAME  HIREDATE    DISTANCE_TO_MILLER NEXT_HD
------ ------ ----------- ------------------ -----------
    10 CLARK  09-JUN-1981                  1 17-NOV-1981
    10 ant    17-NOV-1981                  5 23-JAN-1982
    10 joe    17-NOV-1981                  4 23-JAN-1982
    10 jim    17-NOV-1981                  3 23-JAN-1982
    10 choi   17-NOV-1981                  2 23-JAN-1982
    10 KING   17-NOV-1981                  1 23-JAN-1982
    10 MILLER 23-JAN-1982                  1 (null)
```

如上所示，通过传递适当的距离值以跳过若干行重复数据，LEAD 函数就能实现正确的日期比较了。

第 9 章

日期处理

本章介绍与日期检索和修改相关的实例。日期查询是非常常见的。因此，我们需要知道日期处理的基本思路，并深入理解各种关系数据库管理系统提供的日期处理函数。本章的实例能为我们未来的工作打下坚实的基础，帮助我们处理更为复杂的日期和时间查询。

在开始讲述这些实例之前，我想再强调一次如何使用本书提供的解决方案解决实际问题（我在前言部分也提到过这一点）。我希望你能有"全局视角"。例如，某个实例解决了针对当前月份的一个问题，那么我们其实可以把同样的思路运用在任何一个月份上（需要做一些细微调整），而不是局限于实例里选择的那个月份。我希望你能把本书提供的实例当作解题指南，而不是唯一正确的答案。我没有办法在本书中囊括全部问题的答案，但是如果深入理解了书中给出的解决方案，那么在此基础上加以变通，使之适用于具体问题应该不是那么困难。我也希望你能认真思考一下，除了书中提供的解决方案，是否还有其他替代方案。例如，假设我在某个解决方案里用到了一种关系数据库管理系统的专有函数，那么花一点时间和精力想一想是否还有其他替代方案也是值得的。相较于本书中提供的解决方案，新的办法或许更高效，也可能显得稍微笨一点。无论如何，知道了手头有哪些可选方案有助于我们成为更好的 SQL 程序员。



本章的实例只会用到基本的日期数据类型。如果你需要使用更复杂的日期数据类型，则需要在本章解决方案的基础上做出适当的调整。

9.1　判断闰年

1. 问题

判断当前年份是否闰年。

2. 解决方案

如果你已经有了一段时间的 SQL 编程经验，那么你肯定知道本问题有多种解法。我也尝试过多种解决方案，它们都能给出正确答案，但是本实例提供的方案可能是最为简单的。下面的解决方案仅仅检查 2 月的最后一天：如果有 2 月 29 日，则当前年份是闰年。

DB2

使用 WITH 子句递归查询返回 2 月的每一天，然后调用聚合函数 MAX 确认 2 月的最后一天。

```
 1  with x (dy,mth)
 2    as (
 3  select dy, month(dy)
 4    from (
 5  select (current_date -
 6           dayofyear(current_date) days +1 days)
 7            +1 months as dy
 8    from t1
 9          ) tmp1
10   union all
11  select dy+1 days, mth
12    from x
13   where month(dy+1 day) = mth
14  )
15  select max(day(dy))
16    from x
```

Oracle

使用 LAST_DAY 函数找出 2 月的最后一天。

```
1 select to_char(
2          last_day(add_months(trunc(sysdate,'y'),1)),
3         'DD')
4   from t1
```

PostgreSQL

使用 GENERATE_SERIES 函数返回 2 月的每一天，然后调用聚合函数 MAX 找出 2 月的最后一天。

```
 1  select max(to_char(tmp2.dy+x.id,'DD')) as dy
 2    from (
 3  select dy, to_char(dy,'MM') as mth
 4    from (
 5  select cast(cast(
 6              date_trunc('year',current_date) as date)
 7                         + interval '1 month' as date) as dy
 8    from t1
 9          ) tmp1
10          ) tmp2, generate_series (0,29) x(id)
11    where to_char(tmp2.dy+x.id,'MM') = tmp2.mth
```

MySQL

使用 LAST_DAY 函数找出 2 月的最后一天。

```
1 select day(
```

```
2         last_day(
3         date_add(
4         date_add(
5         date_add(current_date,
6                  interval -dayofyear(current_date) day),
7                  interval 1 day),
8                  interval 1 month))) dy
9   from t1
```

SQL Server

使用 WITH 递归查询返回 2 月的每一天，然后调用聚合函数 MAX 确认 2 月的最后一天。

```
 1   with x (dy,mth)
 2     as (
 3  select dy, month(dy)
 4    from (
 5  select dateadd(mm,1,(getdate()-datepart(dy,getdate()))+1) dy
 6    from t1
 7         ) tmp1
 8   union all
 9  select dateadd(dd,1,dy), mth
10    from x
11   where month(dateadd(dd,1,dy)) = mth
12  )
13  select max(day(dy))
14    from x
```

3. 讨论

DB2

递归视图 X 里的内嵌视图 TMP1 按照下面的步骤返回 2 月的第一天。

(1) 从当前日期开始；

(2) 调用 DAYOFYEAR 函数确认当前日期是当前年份的第几天；

(3) 从当前日期里减去上述步骤算出的那个数字以得到上一年的 12 月 31 日，然后加上 1 天得到当前年份的 1 月 1 日；

(4) 再加上 1 个月得到 2 月 1 日。

上述步骤的运算结果如下所示。

```
select (current_date -
          dayofyear(current_date) days +1 days) +1 months as dy
  from t1

DY
-----------
01-FEB-2005
```

然后，调用 MONTH 函数找出内嵌视图 TMP1 返回的日期对应的月份。

```
select dy, month(dy) as mth
  from (
select (current_date -
          dayofyear(current_date) days +1 days) +1 months as dy
```

```
  from t1
       ) tmp1

DY           MTH
-----------  ---
01-FEB-2005    2
```

到此为止的结果只是作为生成 2 月的每一天的递归操作的起点。为了获得 2 月的每一天，不断为 DY 加上 1 天，直到月份不再是 2 月为止。该 WITH 计算的部分结果如下所示。

```
  with x (dy,mth)
    as (
select dy, month(dy)
  from (
select (current_date -
         dayofyear(current_date) days +1 days) +1 months as dy
  from t1
       ) tmp1
 union all
select dy+1 days, mth
  from x
 where month(dy+1 day) = mth
)
select dy,mth
  from x

DY           MTH
-----------  ---
01-FEB-2005    2
...
10-FEB-2005    2
...
28-FEB-2005    2
```

最后，针对 DY 列调用 MAX 函数返回 2 月的最后一天；如果是 29 日的话，则当前年份是闰年。

Oracle

首先，调用 TRUNC 函数找出当前年份的第一天。

```
select trunc(sysdate,'y')
  from t1

DY
-----------
01-JAN-2005
```

由于 1 月 1 日是一年中的第一天，下一步就是在此基础上加上 1 个月得到 2 月 1 日。

```
select add_months(trunc(sysdate,'y'),1) dy
  from t1

DY
-----------
01-FEB-2005
```

然后，调用 LAST_DAY 找出 2 月的最后一天。

```
select last_day(add_months(trunc(sysdate,'y'),1)) dy
  from t1

DY
-----------
28-FEB-2005
```

最后，调用 TO_CHAR 得到 28 或者 29（这一步不是必需的）。

PostgreSQL

首先观察内嵌视图 TMP1 返回的结果。调用 DATE_TRUNC 函数得到当前年份的第一天，并将其转换为 DATE 类型。

```
select cast(date_trunc('year',current_date) as date) as dy
  from t1

DY
-----------
01-JAN-2005
```

然后，在当前年份第一天的基础上加上 1 个月，得到 2 月的第一天，并转换为 DATE 类型。

```
select cast(cast(
            date_trunc('year',current_date) as date)
                       + interval '1 month' as date) as dy
  from t1

DY
-----------
01-FEB-2005
```

接着，从内嵌视图 TMP1 里返回 DY，并依据 DY 计算出月份的值。调用 TO_CHAR 函数返回月份的值。

```
select dy, to_char(dy,'MM') as mth
   from (
 select cast(cast(
            date_trunc('year',current_date) as date)
                       + interval '1 month' as date) as dy
   from t1
        ) tmp1

DY          MTH
----------- ---
01-FEB-2005   2
```

到此为止的计算结果构成了内嵌视图 TMP2 的结果集。下一步要用到一个非常有用的函数 GENERATE_SERIES 来生成 29 行数据（值从 1 逐一递增到 29）。GENERATE_SERIES 函数返回的每一行（别名为 X）都和内嵌视图 TMP2 的 DY 相加。部分结果如下所示。

```
select tmp2.dy+x.id as dy, tmp2.mth
  from (
```

```
select dy, to_char(dy,'MM') as mth
  from (
select cast(cast(
            date_trunc('year',current_date) as date)
                       + interval '1 month' as date) as dy
  from t1
       ) tmp1
       ) tmp2, generate_series (0,29) x(id)
 where to_char(tmp2.dy+x.id,'MM') = tmp2.mth

DY             MTH
-----------    ---
01-FEB-2005     02
...
10-FEB-2005     02
...
28-FEB-2005     02
```

最后，调用 MAX 函数找出 2 月的最后一天。针对该日期值调用 TO_CHAR 函数将得到 28 或者 29。

MySQL

首先找出当前年份的第一天：先计算出当前日期是当前年份的第几天，用当前日期减去该值，然后再加上 1 天。DATE_ADD 函数能完成这一步。

```
select date_add(
       date_add(current_date,
                interval -dayofyear(current_date) day),
                interval 1 day) dy
  from t1

DY
-----------
01-JAN-2005
```

接着，再次调用 DATE_ADD 函数在上述计算结果的基础上加上 1 个月。

```
select date_add(
       date_add(
       date_add(current_date,
                interval -dayofyear(current_date) day),
                interval 1 day),
                interval 1 month) dy
  from t1

DY
-----------
01-FEB-2005
```

现在得到了 2 月 1 日的日期值，接着调用 LAST_DAY 函数找出 2 月的最后一天。

```
select last_day(
       date_add(
       date_add(
       date_add(current_date,
```

```
                    interval -dayofyear(current_date) day),
                    interval 1 day),
                    interval 1 month)) dy
  from t1

DY
-----------
28-FEB-2005
```

最后，调用 DAY 函数返回 28 或者 29（这一步不是必需的）。

SQL Server

该解决方案利用 WITH 递归查询生成 2 月的每一天。第一步先找出 2 月的第一天。为达到此目的，先找出当前年份的第一天：计算出当前日期是当前年份的第几天，用当前日期减去该值，然后再加上 1 天。既然有了当前年份的第一天，调用 DATEADD 函数加上 1 个月，就能得到 2 月的第一天了。

```
select dateadd(mm,1,(getdate()-datepart(dy,getdate()))+1) dy
  from t1

DY
-----------
01-FEB-2005
```

接着，返回 2 月的第一天，并计算出该日期对应月份的数值形式。

```
select dy, month(dy) mth
  from (
select dateadd(mm,1,(getdate()-datepart(dy,getdate()))+1) dy
  from t1
       ) tmp1

DY            MTH
-----------   ---
01-FEB-2005     2
```

然后利用 WITH 子句的递归特性，不断为内嵌视图 TMP1 返回的 DY 加上 1，直到日期对应的月份不再是 2 月，部分结果如下所示。

```
  with x (dy,mth)
    as (
select dy, month(dy)
  from (
select dateadd(mm,1,(getdate()-datepart(dy,getdate()))+1) dy
  from t1
       ) tmp1
 union all
select dateadd(dd,1,dy), mth
  from x
 where month(dateadd(dd,1,dy)) = mth
)
select dy,mth from x

DY            MTH
```

```
-----------  ---
01-FEB-2005   02
...
10-FEB-2005   02
...
28-FEB-2005   02
```

现在得到了 2 月的每一天，最后调用 MAX 函数看一下最后一天是 28 日还是 29 日。还可以调用 DAY 函数返回数字 28 或者 29，而不是一个日期值；不过，这一步不是必需的。

9.2 计算一年有多少天

1. 问题

计算当前年份有多少天。

2. 解决方案

计算当前年份有多少天，等同于计算下一年的第一天和当前年份的第一天之间的差值（以天为单位）。对于下面的所有解决方案，求解步骤都如下所示。

(1) 找到当前年份的第一天；
(2) 在上述结果的基础上加上 1 年（以得到下一年的第一天）；
(3) 用第 2 步得到的结果减去第一步得到的结果。

下面的各种解决方案的不同之处仅在于上述各步骤使用的内置函数不一样。

DB2

使用 DAYOFYEAR 函数找出当前年份的第一天，并使用 DAYS 函数得出当前年份有多少天。

```
1 select days((curr_year + 1 year)) - days(curr_year)
2   from (
3 select (current_date -
4         dayofyear(current_date) day +
5          1 day) curr_year
6   from t1
7        ) x
```

Oracle

使用 TRUNC 函数找出当前年份的第一天，并调用 ADD_MONTHS 函数得到下一年的第一天。

```
1 select add_months(trunc(sysdate,'y'),12) - trunc(sysdate,'y')
2   from dual
```

PostgreSQL

使用 DATE_TRUNC 函数找出当前年份的第一天，然后借助 INTERVAL 关键字计算出下一年的第一天。

```
1 select cast((curr_year + interval '1 year') as date) - curr_year
2   from (
3 select cast(date_trunc('year',current_date) as date) as curr_year
4   from t1
5        ) x
```

MySQL

使用 ADDDATE 函数找出当前年份的第一天。调用 DATEDIFF 函数，并借助 INTERVAL 关键字计算出当前年份有多少天。

```
1 select datediff((curr_year + interval 1 year),curr_year)
2   from (
3 select adddate(current_date,-dayofyear(current_date)+1) curr_year
4   from t1
5        ) x
```

SQL Server

使用 DATEADD 函数找出当前年份的第一天。调用 DATEDIFF 函数计算出当前年份有多少天。

```
1 select datediff(d,curr_year,dateadd(yy,1,curr_year))
2   from (
3 select dateadd(d,-datepart(dy,getdate())+1,getdate()) curr_year
4   from t1
5        ) x
```

3. 讨论

DB2

首先找出当前年份的第一天。调用 DAYOFYEAR 函数计算出当前日期是当前年份的第几天，用当前日期减去该值就能得到上一年的最后一天，然后加上 1 天。

```
select (current_date -
        dayofyear(current_date) day +
         1 day) curr_year
  from t1

CURR_YEAR
-----------
01-JAN-2005
```

现在得到了当前年份的第一天，只要在此基础上加上 1 年，就能得到下一年的第一天。然后，用下一年的第一天减去当前年份的第一天，就能得到答案了。

Oracle

首先找出当前年份的第一天，直接调用内置函数 TRUNC 并把 Y 作为第二个参数（因而会截断当前系统日期值得到当前年份的第一天）即可。

```
select select trunc(sysdate,'y') curr_year
  from dual

CURR_YEAR
-----------
01-JAN-2005
```

然后，在上述计算结果的基础上加上 1 年得到下一年的第一天。最后，两个日期相减得到当前年份有多少天。

PostgreSQL

先找到当前年份的第一天。为此，要调用 DATE_TRUNC 函数，如下所示。

```
select cast(date_trunc('year',current_date) as date) as curr_year
  from t1

CURR_YEAR
-----------
01-JAN-2005
```

然后，在上述计算结果的基础上简单地加上 1 年，计算出下一年的第一天。接着，只需要把两个日期相减。要记得用靠后的日期减去较早的日期。得到的结果就是当前年份有多少天。

MySQL

第一步是找出当前年份的第一天。调用 DAYOFYEAR 函数得到当前日期是当前年份的第几天。用当前日期减去该值，然后加上 1 天。

```
select adddate(current_date,-dayofyear(current_date)+1) curr_year
  from t1

CURR_YEAR
-----------
01-JAN-2005
```

现在得到了当前年份的第一天，下一步是在此基础上加上 1 年得到下一年的第一天。然后，用下一年的第一天减去当前年份的第一天。得到的结果就是当前年份有多少天。

SQL Server

第一步是找出当前年份的第一天。调用 DATEADD 和 DATEPART 函数从当前日期减去当前年份已经过去的天数，然后再加上 1 天。

```
select dateadd(d,-datepart(dy,getdate())+1,getdate()) curr_year
  from t1

CURR_YEAR
-----------
01-JAN-2005
```

现在得到了当前年份的第一天，下一步是在此基础上加上 1 年得到下一年的第一天。然后，用下一年的第一天减去当前年份的第一天，得到的结果就是当前年份的有多少天。

9.3　从给定日期值里提取年月日时分秒

1. 问题

把当前日期值分解为六个部分：年、月、日、时、分和秒，并且希望结果以数字的形式返回。

2. 解决方案

这里以当前日期为例。但是，本实例适用于任何其他日期值。在第 1 章，我提到过学习和利用数据库内置函数的重要性。对于日期处理而言，这一点尤其重要。除了本实例提供的做法，还有其他方式能够从给定日期值里提取年、月、日、时、分、秒等时间单位。总

之，多尝试不同的做法和技巧是非常有益的。

DB2

DB2 实现了一组内置函数帮助我们方便地提取出一个日期值的每一个组成部分。这些函数分别被命名为 HOUR、MINUTE、SECOND、DAY、MONTH 和 YEAR，直观地表明它们要返回的时间单位。如果我们想要“天”，那就调用 DAY 函数；如果想要“小时”，那就调用 HOUR 函数，等等。示例如下。

```
1 select   hour( current_timestamp ) hr,
2        minute( current_timestamp ) min,
3        second( current_timestamp ) sec,
4           day( current_timestamp ) dy,
5         month( current_timestamp ) mth,
6          year( current_timestamp ) yr
7   from t1

  HR   MIN   SEC    DY   MTH    YR
---- ----- ----- ----- ----- -----
  20    28    36    15     6  2005
```

Oracle

使用 TO_CHAR 和 TO_NUMBER 函数从一个日期值里提取各种时间单位。

```
1  select to_number(to_char(sysdate,'hh24')) hour,
2         to_number(to_char(sysdate,'mi')) min,
3         to_number(to_char(sysdate,'ss')) sec,
4         to_number(to_char(sysdate,'dd')) day,
5         to_number(to_char(sysdate,'mm')) mth,
6         to_number(to_char(sysdate,'yyyy')) year
7    from dual

HOUR   MIN   SEC    DY   MTH  YEAR
---- ----- ----- ----- ----- -----
  20    28    36    15     6  2005
```

PostgreSQL

使用 TO_CHAR 和 TO_NUMBER 函数从一个日期值里提取各种时间单位。

```
1 select to_number(to_char(current_timestamp,'hh24'),'99') as hr,
2        to_number(to_char(current_timestamp,'mi'),'99') as min,
3        to_number(to_char(current_timestamp,'ss'),'99') as sec,
4        to_number(to_char(current_timestamp,'dd'),'99') as day,
5        to_number(to_char(current_timestamp,'mm'),'99') as mth,
6        to_number(to_char(current_timestamp,'yyyy'),'9999') as yr
7   from t1

  HR   MIN   SEC    DY   MTH    YR
---- ----- ----- ----- ----- -----
  20    28    36    15     6  2005
```

MySQL

使用 DATE_FORMAT 函数从一个日期值里提取各种时间单位。

```
1 select date_format(current_timestamp,'%k') hr,
2        date_format(current_timestamp,'%i') min,
3        date_format(current_timestamp,'%s') sec,
4        date_format(current_timestamp,'%d') dy,
5        date_format(current_timestamp,'%m') mon,
6        date_format(current_timestamp,'%Y') yr
7   from t1

  HR   MIN   SEC    DY   MTH    YR
---- ----- ----- ----- ----- -----
  20    28    36    15     6  2005
```

SQL Server

使用 DATEPART 函数从一个日期值里提取各种时间单位。

```
1 select datepart( hour,  getdate()) hr,
2        datepart( minute,getdate()) min,
3        datepart( second,getdate()) sec,
4        datepart( day,   getdate()) dy,
5        datepart( month, getdate()) mon,
6        datepart( year,  getdate()) yr
7   from t1

  HR   MIN   SEC    DY   MTH    YR
---- ----- ----- ----- ----- -----
  20    28    36    15     6  2005
```

3. 讨论

以上这些解决方案并没有什么特别之处，只是尽量利用了数据库中的内置函数。我们应该花一些时间去学习这些与日期相关的函数。不过，本实例的各个解决方案仅仅展示了这些函数的部分功能。如果仔细研究的话，我们会看到每一个函数都能接受更多的参数，并能返回更多信息，只是我们无法在本实例里一一演示给你看。

9.4　计算一个月的第一天和最后一天

1. 问题

你希望知道当前月份的第一天和最后一天。

2. 解决方案

这里给出的解决方案是找出当前月份的第一天和最后一天。其实，并非一定要选择当前月份。略微改动一下的话，本方案就能适用于任何月份。

DB2

使用 DAY 函数计算出当前日期是当前月份的第几天。用当前日期减去该值，并加上 1，就得到了当前月份的第一天。为了获取当前月份的最后一天，再次针对当前日期调用 DAY 函数，然后在当前日期的基础上加上 1 个月，并减去上述 DAY 函数调用的返回值。

```
1 select (current_date - day(current_date) day +1 day) firstday,
2        (current_date +1 month -day(current_date) day) lastday
3   from t1
```

Oracle

使用 TRUNC 函数找出当前月份的第一天，并使用 LAST_DAY 函数找出当前月份的最后一天。

```
1 select trunc(sysdate,'mm') firstday,
2        last_day(sysdate) lastday
3   from dual
```



上述调用 TRUNC 函数会丢掉日期值里原本包含的时间部分，而 LAST_DAY 函数则会保留时间值。

PostgreSQL

针对当前日期调用 DATE_TRUNC 函数得到当前月份的第一天。既然知道了当前月份的第一天，先加上 1 个月，再减去 1 天，这样就得到了当前月份的最后一天。

```
1 select firstday,
2        cast(firstday + interval '1 month'
3                      - interval '1 day' as date) as lastday
4   from (
5 select cast(date_trunc('month',current_date) as date) as firstday
6   from t1
7        ) x
```

MySQL

使用 DATE_ADD 和 DAY 函数计算出当前日期是当前月份的第几天。然后从当前日期里减去该计算结果，并加 1，就得到了当前月份的第一天。为了得到当前月份的最后一天，则使用 LAST_DAY 函数。

```
1 select date_add(current_date,
2                 interval -day(current_date)+1 day) firstday,
3        last_day(current_date) lastday
4   from t1
```

SQL Server

使用 DATEADD 和 DAY 函数计算出当前日期是当前月份的第几天。然后从当前日期里减去该计算结果，并加 1，就得到了当前月份的第一天。为得到当前月份最后一天，再次针对当前日期调用 DAY 函数，然后再次调用 DATEADD 函数在当前日期的基础上加上 1 个月，并减去上述调用 DAY 函数的返回值。

```
1 select dateadd(day,-day(getdate())+1,getdate()) firstday,
2        dateadd(day,
3                -day(getdate( )),
4                dateadd(month,1,getdate())) lastday
5   from t1
```

3. 讨论

DB2

为了得到一个月的第一天，调用 DAY 函数。有了 DAY 函数，就能方便地知道给定日期是当前月份的第几天。如果用当前日期减去 DAY(CURRENT_DATE) 函数调用的返回值，我们将得到上个月的最后一天；在此基础上加上 1 天的话，就能算出当前月份的第一天。为找

出一个月的最后一天，要先在当前日期的基础上加上 1 个月。这将得到下个月的同一天（即使接下来的一个月天数少于当前月份，该数学计算仍将返回正确的结果）。然后减去 DAY(CURRENT_DATE) 函数调用的返回值，就得到了当前月份的最后一天。

Oracle

为了得到当前月份的第一天，调用 TRUNC 函数，并把 mm 作为第二个参数，这样就能“截断”当前日期得到当前月份的第一天。为获取当前月份的最后一天，只需调用 LAST_DAY 函数即可。

PostgreSQL

为得到当前月份的第一天，调用 DATE_TRUNC 函数，并把 month 作为第二个参数，这样就能“截断”当前日期得到当前月份的第一天。为获取当前月份的最后一天，在当前月份第一天的基础上加上 1 个月，然后再减去 1 天即可。

MySQL

为了得到当前月份的第一天，调用 DAY 函数。有了 DAY 函数，就能方便地知道给定日期是当前月份的第几天。如果用当前日期减去 DAY(CURRENT_DATE) 函数调用的返回值，我们将得到上个月的最后一天；在此基础上加上 1 天的话，就能算出当前月份的第一天。为获取当前月份的最后一天，只要调用 LAST_DAY 函数即可。

SQL Server

为了得到一个月的第一天，调用 DAY 函数。有了 DAY 函数，就能方便地知道给定日期是当前月份的第几天。如果用当前日期减去 DAY(GETDATE( )) 函数调用的返回值，我们将得到上个月的最后一天；在此基础上加上 1 天的话，就能计算出当前月份的第一天。为了获取当前月份的最后一天，需要调用 DATEADD 函数。在当前日期的基础上加上 1 个月，然后减去调用 DAY(GETDATE( )) 函数的返回值，这样就得到了当前月份的最后一天。

9.5　列出一年中所有的星期五

1. 问题

对于一周的某一天，你想找出一年中与之对应的所有日期。例如，你希望生成一个列表，列出当前年份所有的星期五。

2. 解决方案

不论使用哪一种数据库，解决本问题的关键都在于先列出当前年份的每一天，然后筛选出符合条件的日期。下面的解决方案以找出所有的星期五为例。

DB2

使用 WITH 递归查询列出当前年份的每一天，然后调用 DAYNAME 函数筛选出星期五对应的日期。

```
1   with x (dy,yr)
2     as (
3  select dy, year(dy) yr
4    from (
5  select (current_date -
```

```
6           dayofyear(current_date) days +1 days) as dy
7    from t1
8          ) tmp1
9   union all
10  select dy+1 days, yr
11    from x
12   where year(dy +1 day) = yr
13  )
14  select dy
15    from x
16   where dayname(dy) = 'Friday'
```

Oracle

使用 CONNECT BY 递归查询列出当前年份的每一天，然后调用 TO_CHAR 函数筛选出星期五对应的日期。

```
1    with x
2      as (
3   select trunc(sysdate,'y')+level-1 dy
4     from t1
5     connect by level <=
6        add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
7   )
8   select *
9     from x
10   where to_char( dy, 'dy') = 'fri'
```

PostgreSQL

使用 GENERATE_SERIES 函数列出当前年份的每一天，然后调用 TO_CHAR 函数筛选出星期五对应的日期。

```
1 select cast(date_trunc('year',current_date) as date)
2         + x.id as dy
3   from generate_series (
4         0,
5         ( select cast(
6                 cast(
7            date_trunc('year',current_date) as date)
8                       + interval '1 years' as date)
9                       - cast(
10                   date_trunc('year',current_date) as date) )-1
11        ) x(id)
12  where to_char(
13          cast(
14     date_trunc('year',current_date)
15               as date)+x.id,'dy') = 'fri'
```

MySQL

使用数据透视表 T500 列出当前年份的每一天，然后调用 DAYNAME 函数筛选出星期五对应的日期。

```
1   select dy
2     from (
```

```
 3  select adddate(x.dy,interval t500.id-1 day) dy
 4    from (
 5  select dy, year(dy) yr
 6    from (
 7  select adddate(
 8         adddate(current_date,
 9                 interval -dayofyear(current_date) day),
10                 interval 1 day ) dy
11    from t1
12         ) tmp1
13         ) x,
14         t500
15   where year(adddate(x.dy,interval t500.id-1 day)) = x.yr
16         ) tmp2
17   where dayname(dy) = 'Friday'
```

SQL Server

使用 WITH 递归查询列出当前年份的每一天，然后调用 DAYNAME 函数筛选出星期五对应的日期。

```
 1    with x (dy,yr)
 2      as (
 3  select dy, year(dy) yr
 4    from (
 5  select getdate()-datepart(dy,getdate())+1 dy
 6     from t1
 7          ) tmp1
 8   union all
 9  select dateadd(dd,1,dy), yr
10   from x
11   where year(dateadd(dd,1,dy)) = yr
12  )
13  select x.dy
14    from x
15   where datename(dw,x.dy) = 'Friday'
16  option (maxrecursion 400)
```

3. 讨论

DB2

为了找出当前年份所有的星期五，我们必须先列出来当前年份的每一天。第一步要调用 DAYOFYEAR 函数找到当前年份的第一天。从当前日期里减去 DAYOFYEAR(CURRENT_DATE) 函数调用的返回值可以得到上一年 12 月 31 日，再加上 1 天就得到了当前年份的第一天。

```
select (current_date -
         dayofyear(current_date) days +1 days) as dy
  from t1

DY
-----------
01-JAN-2005
```

现在我们知道了当前年份的第一天，接着使用 WITH 子句在当前年份第一天的基础上逐次加上 1 天，直至得到的日期不再属于当前年份。上述得到的结果集将是当前年份的每一天

（递归视图 X 查询的部分结果如下所示）。

```
  with x (dy,yr)
    as (
select dy, year(dy) yr
  from (
select (current_date -
         dayofyear(current_date) days +1 days) as dy
  from t1
       ) tmp1
union all
select dy+1 days, yr
  from x
 where year(dy +1 day) = yr
)
select dy
  from x

DY
-----------
01-JAN-2005
...
15-FEB-2005
...
22-NOV-2005
...
31-DEC-2005
```

最后，调用 DAYNAME 函数筛选出星期五对应的日期。

Oracle

为了找到当前年份所有的星期五，我们必须先列出来当前年份的每一天。首先调用 TRUNC 函数得到当前年份的第一天。

```
select trunc(sysdate,'y') dy
  from t1

DY
-----------
01-JAN-2005
```

然后，使用 CONNECT BY 子句返回当前年份的每一天（参见 第 13 章中的相关内容，以了解如何使用 CONNECT BY 生成行数据）。



顺便说一下，虽然本实例采用了基于 WITH 子句的解决方案，但其实也可以使用内嵌视图达到同样目的。

在写作本书时，Oracle 的 WITH 子句并不用于实现递归操作（这不同于 DB2 和 SQL Server），递归操作是由 CONNECT BY 完成的。视图 X 返回的部分结果集如下所示。

```
  with x
    as (
select trunc(sysdate,'y')+level-1 dy
```

```
  from t1
  connect by level <=
     add_months(trunc(sysdate,'y'),12)-trunc(sysdate,'y')
)
select *
  from x

DY
-----------
01-JAN-2005
...
15-FEB-2005
...
22-NOV-2005
...
31-DEC-2005
```

最后，调用 TO_CHAR 函数筛选出星期五对应的日期。

PostgreSQL

为了找出当前年份所有的星期五，我们必须先把当前年份的每一天都当作一条记录返回。这里需要使用 GENERATE_SERIES 函数。GENERATE_SERIES 函数返回的起点和终点值分别是 0 和当前年份总天数减 1。传递给 GENERATE_SERIES 函数的第一个参数是 0，第二个参数则是一个查询，该查询用于计算出当前年份有多少天。（因为我们要在当前年份第一天的基础上逐日累加，实际上要累加的天数恰好比当前年份的总天数少 1 天，这样才不至于溢出到下一年。）GENERATE_SERIES 函数的第二个参数返回的结果如下所示。

```
select cast(
       cast(
 date_trunc('year',current_date) as date)
           + interval '1 years' as date)
           - cast(
        date_trunc('year',current_date) as date)-1 as cnt
  from t1

CNT
---
364
```

请记住，根据上述结果集，FROM 子句里的 GENERATE_SERIES 函数调用看起来是这样的：GENERATE_SERIES(0,364)。如果是闰年，例如 2004 年，第二个参数则是 365。

为了生成当前年份全部日期的列表，下一步就要把 GENERATE_SERIES 函数的返回值依次加上当前年份的第一天。部分结果如下所示。

```
select cast(date_trunc('year',current_date) as date)
       + x.id as dy
  from generate_series (
        0,
        ( select cast(
                 cast(
           date_trunc('year',current_date) as date)
                      + interval '1 years' as date)
```

```
                      - cast(
                  date_trunc('year',current_date) as date) )-1
       ) x(id)

DY
----------
01-JAN-2005
...
15-FEB-2005
...
22-NOV-2005
...
31-DEC-2005
```

最后，调用 TO_CHAR 函数筛选出星期五对应的日期。

MySQL

为了找出当前年份的全部星期五，我们先列出来当前年份的每一天。首先要调用 DAYOFYEAR 函数找出当前年份的第一天。从当前日期里减去 DAYOFYEAR(CURRENT_DATE) 函数调用的返回值，然后再加上 1 天，这样就得到了当前年份的第一天。

```
select adddate(
       adddate(current_date,
               interval -dayofyear(current_date) day),
               interval 1 day ) dy
  from t1

DY
-----------
01-JAN-2005
```

然后，使用 T500 表生成足够多的行以返回当前年份的每一天。我们要在当前年份第一天的基础上逐一加上 T500.ID 的值，直至当前年份结束。这个操作的部分结果如下。

```
select adddate(x.dy,interval t500.id-1 day) dy
  from (
select dy, year(dy) yr
  from (
select adddate(
       adddate(current_date,
               interval -dayofyear(current_date) day),
               interval 1 day ) dy
  from t1
       ) tmp1
       ) x,
       t500
 where year(adddate(x.dy,interval t500.id-1 day)) = x.yr

DY
----------
01-JAN-2005
...
15-FEB-2005
...
```

```
22-NOV-2005
...
31-DEC-2005
```

最后，调用 DAYNAME 函数筛选出星期五对应的日期。

SQL Server

为了找出当前年份所有的星期五，我们必须先列出当前年份的每一天。首先要调用 DATEPART 函数得到当前年份的第一天。从当前日期里减去 DATEPART(DY,GETDATE()) 函数调用的返回值，并加上 1 天，就得到了当前年份的第一天。

```
select getdate()-datepart(dy,getdate())+1 dy
  from t1

DY
-----------
01-JAN-2005
```

现在知道了当前年份的第一天，接着使用 WITH 子句和 DATEADD 函数在第一天的基础上逐次加上 1 天，直至当前年份的最后一天。这样一来，得到的结果集就是当前年份的每一天（递归视图 X 返回的部分行如下所示）。

```
with x (dy,yr)
  as (
select dy, year(dy) yr
  from (
select getdate()-datepart(dy,getdate())+1 dy
  from t1
       ) tmp1
 union all
select dateadd(dd,1,dy), yr
  from x
 where year(dateadd(dd,1,dy)) = yr
)
select x.dy
  from x
option (maxrecursion 400)

DY
-----------
01-JAN-2005
...
15-FEB-2005
...
22-NOV-2005
...
31-DEC-2005
```

最后，调用 DATENAME 函数筛选出星期五对应的日期。对于本解决方案而言，我们必须设置 MAXRECURSION 的值，使之不小于 366（这是为了过滤递归视图 X，使得查询结果都是当前年份的数据，并保证结果集不超过 366 行）。

9.6　找出当前月份的第一个和最后一个星期一

1. 问题

例如，你希望找出当前月份的第一个和最后一个星期一。

2. 解决方案

在这里我们选择当前月份和星期一。事实上，下面给出的解决方案适用于任何一个月份和一周七天里的任何一天。一周有七天，一旦我们知道了第 1 个星期一对应的日期，那么加上 7 天就能得到第 2 个星期一，加上 14 天就能得到第 3 个星期一。类似地，如果我们知道当前月份最后一个星期一对应的日期，那么减去 7 天就能得到第 3 个星期一，减去 14 天就能得到第 2 个星期一。

DB2

使用 WITH 递归查询生成当前月份的每一天，并使用 CASE 表达式标记所有的星期一。第一个和最后一个星期一分别是最早的和最晚的、带有标记的日期。

```
1    with x (dy,mth,is_monday)
2      as (
3  select dy,month(dy),
4         case when dayname(dy)='Monday'
5              then 1 else 0
6         end
7    from (
8  select (current_date-day(current_date) day +1 day) dy
9    from t1
10          ) tmp1
11  union all
12 select (dy +1 day), mth,
13        case when dayname(dy +1 day)='Monday'
14              then 1 else 0
15        end
16   from x
17  where month(dy +1 day) = mth
18 )
19 select min(dy) first_monday, max(dy) last_monday
20   from x
21  where is_monday = 1
```

Oracle

使用 NEXT_DAY 和 LAST_DAY 函数，辅以少许日期计算的技巧，以找出当前月份的第一个和最后一个星期一。

```
select next_day(trunc(sysdate,'mm')-1,'MONDAY') first_monday,
       next_day(last_day(trunc(sysdate,'mm'))-7,'MONDAY') last_monday
  from dual
```

PostgreSQL

使用 DATE_TRUNC 函数找出当前月份的第一天。有了当前月份第一天的日期，就能通过简单的数学运算（星期日到星期六分别对应数值 1 和 7）得到当前月份的第一个和最后一个星期一。

```
1  select first_monday,
2         case to_char(first_monday+28,'mm')
3              when mth then first_monday+28
4                       else first_monday+21
5         end as last_monday
6    from (
7  select case sign(cast(to_char(dy,'d') as integer)-2)
8              when  0
9              then dy
10             when -1
11             then dy+abs(cast(to_char(dy,'d') as integer)-2)
12             when 1
13             then (7-(cast(to_char(dy,'d') as integer)-2))+dy
14        end as first_monday,
15        mth
16   from (
17 select cast(date_trunc('month',current_date) as date) as dy,
18        to_char(current_date,'mm') as mth
19   from t1
20        ) x
21        ) y
```

MySQL

使用 ADDDATE 函数找出当前月份的第一天。有了当前月份第一天的日期，就能通过简单的数学运算（星期日到星期六分别对应数值 1 和 7）得到当前月份的第一个和最后一个星期一。

```
1  select first_monday,
2         case month(adddate(first_monday,28))
3              when mth then adddate(first_monday,28)
4                       else adddate(first_monday,21)
5         end last_monday
6    from (
7  select case sign(dayofweek(dy)-2)
8              when  0 then dy
9              when -1 then adddate(dy,abs(dayofweek(dy)-2))
10             when  1 then adddate(dy,(7-(dayofweek(dy)-2)))
11        end first_monday,
12        mth
13   from (
14 select adddate(adddate(current_date,-day(current_date)),1) dy,
15        month(current_date) mth
16   from t1
17        ) x
18        ) y
```

SQL Server

使用 WITH 递归查询生成当前月份的每一天，并使用 CASE 表达式标记所有的星期一。第一个和最后一个星期一分别是最早的和最晚的、带有标记的日期。

```
1   with x (dy,mth,is_monday)
2     as (
3  select dy,mth,
4         case when datepart(dw,dy) = 2
```

```
 5              then 1 else 0
 6         end
 7    from (
 8  select dateadd(day,1,dateadd(day,-day(getdate()),getdate())) dy,
 9         month(getdate()) mth
10    from t1
11         ) tmp1
12   union all
13  select dateadd(day,1,dy),
14         mth,
15         case when datepart(dw,dateadd(day,1,dy)) = 2
16              then 1 else 0
17         end
18    from x
19   where month(dateadd(day,1,dy)) = mth
20  )
21  select min(dy) first_monday,
22         max(dy) last_monday
23    from x
24   where is_monday = 1
```

3. 讨论

DB2 和 SQL Server

DB2 和 SQL Server 的解决方案使用了不同的函数，但其做法并无二致。如果我们仔细审视这两个解决方案的话，就会发现它们的唯一差别在于日期的加法运算。下面的讨论将涵盖这两种数据库，但是会借用 DB2 解决方案的代码来演示中间步骤的结果。



如果你没有办法找到支持 WITH 递归查询语法的 SQL Server 或 DB2 版本，不妨采用 PostgreSQL 解决方案的做法。

为了找出当前月份的第一个和最后一个星期一，第一步是找到当前月份的第一天。递归视图 X 里的内嵌视图 TMP1 可以找出当前月份的第一天，它的做法是先找出当前日期，并且要特别地计算出该日期是当前月份的第几天。计算出当前日期是当前月份的第几天，就知道到该日期为止这个月已经过去了多少天（例如，4 月 10 日是 4 月份的第 10 天）。从当前日期里减去该值，就退回到了上个月的最后一天（例如，4 月 10 日减去 10 天，结果就是 3 月份的最后一天）。做过上述减法运算之后，只要再加上 1 天就能得到当前月份的第一天了。

```
select (current_date-day(current_date) day +1 day) dy
  from t1

DY
-----------
01-JUN-2005
```

下一步要找出当前日期对应的月份，调用 MONTH 函数，并使用简单的 CASE 表达式来确认这个月的第一天是不是星期一。

```
select dy, month(dy) mth,
       case when dayname(dy)='Monday'
```

```
             then 1 else 0
        end is_monday
  from (
select (current_date-day(current_date) day +1 day) dy
  from t1
       ) tmp1

DY          MTH  IS_MONDAY
----------- --- ----------
01-JUN-2005   6          0
```

接着，借助 WITH 子句的递归特性在当前月份第一天的基础上不断地加上 1 天，直到当前月份最后一天。同时，也可以使用 CASE 表达式来确认每一个日期是不是星期一（星期一将被标记为“1”）。递归视图 X 的部分查询结果如下所示。

```
with x (dy,mth,is_monday)
     as (
 select dy,month(dy) mth,
        case when dayname(dy)='Monday'
             then 1 else 0
        end is_monday
   from (
 select (current_date-day(current_date) day +1 day) dy
   from t1
        ) tmp1
  union all
 select (dy +1 day), mth,
        case when dayname(dy +1 day)='Monday'
             then 1 else 0
        end
   from x
  where month(dy +1 day) = mth
 )
select *
  from x

DY          MTH  IS_MONDAY
----------- --- ----------
01-JUN-2005  6           0
02-JUN-2005  6           0
03-JUN-2005  6           0
04-JUN-2005  6           0
05-JUN-2005  6           0
06-JUN-2005  6           1
07-JUN-2005  6           0
08-JUN-2005  6           0
...
```

只有星期一对应的 IS_MONDAY 是 1，因而最后一步是针对 IS_MONDAY 等于 1 的行调用聚合函数 MIN 和 MAX，以找出当前月份的第一个和最后一个星期一。

Oracle

有了 NEXT_DAY 函数，本问题就很容易解决了。为了找出当前月份的第一个星期一，先得找到前一个月的最后一天，这需要借助一些日期计算，包括 TRUNC 函数。

```
select trunc(sysdate,'mm')-1 dy
  from dual

DY
-----------
31-MAY-2005
```

然后，调用 NEXT_DAY 函数计算出紧随前个月最后一天出现的第一个星期一（也就是当前月份的第一个星期一）。

```
select next_day(trunc(sysdate,'mm')-1,'MONDAY') first_monday
  from dual

FIRST_MONDAY
------------
06-JUN-2005
```

为了找出当前月份的最后一个星期一，先要调用 TRUNC 函数计算出当前月份第一天。

```
select trunc(sysdate,'mm') dy
  from dual

DY
-----------
01-JUN-2005
```

下一步是找出这个月的最后一周（最后 7 天），调用 LAST_DAY 函数找到这个月的最后一天，然后减去 7 天。

```
select last_day(trunc(sysdate,'mm'))-7 dy
  from dual

DY
-----------
23-JUN-2005
```

我们之所以要从当前月份的最后一天向前倒退 7 天，是为了保证这 7 天里至少剩下一个星期一。最后，调用 NEXT_DAY 函数找到下一个（当前月份最后一个）星期一。

```
select next_day(last_day(trunc(sysdate,'mm'))-7,'MONDAY') last_monday
  from dual

LAST_MONDAY
-----------
27-JUN-2005
```

PostgreSQL 和 MySQL

PostgreSQL 和 MySQL 解决方案的思路也很类似，差别在于用到的内置函数不同。虽然代码有点长，这两个解决方案的查询语句其实非常简单。而且，在计算当前月份第一个和最后一个星期一的过程中，并未增加多少额外的复杂度。

首先找出当前月份的第一天，紧接着要找出当前月份的第一个星期一。由于没有内置函数可以找到下一个星期一，需要做一些日期运算。（这两个解决方案中任何一个的）第 7 行

开始的 CASE 表达式评估当前月份的第一天是不是星期一。（PostgresSQL 的）TO_CHAR 函数在指定了 D 或者 d 格式的情况下会返回 1 到 7，分别表示星期日到星期六，（MySQL 的）DAYOFWEEK 函数亦然。其中，星期一对应的值始终是 2。 CASE 表达式要评估的对象是 SIGN 函数的返回值，当前月份的第一天对应的数值（不论它是星期几）减去星期一对应的数值 2，该减法运算的结果传递给 SIGN 函数。如果结果是 0，那么当前月份的第一天就是星期一，并且也是当前月份的第一个星期一。如果结果是 –1，那么当前月份的第一天就是星期日，只要在当前月份的第一天的基础上再加上 2 和 1（分别代表星期一和星期日）之间相差的天数就能得到当前月份的第一个星期一了。



如果觉得理解起来有点困难，不妨暂时抛开某天是星期几这种想法，只关注数字运算。例如，如果今天是星期二，而我们想知道下一个星期五的日期。可以调用指定了 d 格式参数的 TO_CHAR 函数，也可以调用 DAYOFWEEK 函数，星期五对应数字 6，星期二则对应 3。从 3 数到 6，直接做减法即可（6–3 = 3），然后加上两者之中较小的那个数字（(6–3) + 3 = 6）。因此，先别管具体的日期是什么，如果起始日期对应的数值小于目标日期，那么在起始日期的基础上加上两个日期的差值就能得到目标日期对应的日期了。

如果结果是 1，那么当前月份的第一天就介于星期二和星期六之间（包含起止日期）。如果当前月份第一天对应的数值大于 2（星期一），先算出当前月份第一天和星期一对应的数字（2）之间的差值，用 7 减去该差值，再将上述计算结果加到当前月份的第一天上。这样，就得到了当前月份第一个星期一。



再次提醒，如果你觉得理解起来有点困难，不妨暂时抛开某天是星期几这种想法，只关注数字运算。例如，假定现在是星期五，我们想找出下个星期二对应的日期。星期二（对应数字 3）比星期五（对应数字 6）小。从 6 数到 3，先计算出两个数字之间差值，并用 7 减去该差值（7−(|3−6|) = 4），然后在起始日星期五的基础上加上这个结果（4）。（“|3−6|”里的竖线表示生成差值对应的绝对值。）这里不能在 6 的基础上再加上 4（这将得到 10），我们需要在星期五的基础上加上 4 天，这样才能得到下一个星期二。

上述 CASE 表达式的思路是，为 PostgreSQL 和 MySQL 实现类似 Oracle 的 NEXT_DAY 函数的功能。如果我们不以当前月份的第一天为起点，DY 的值就变成了 CURRENT_DATE 函数的返回值，而 CASE 表达式的计算结果也就变成了从当前日期开始算起的下一个星期一（如果当前日期是星期一，那么返回值会是其自身）。

现在知道了当前月份的第一个星期一，加上 21 天或者 28 天就能得到当前月份的最后一个星期一。第 2 行到第 5 行之间的 CASE 表达式决定应该加上 21 天还是 28 天，这要视加上 28 天之后会不会落入下个月而定。该 CASE 表达式通过下述步骤实现该计算。

(1) 为 FIRST_MONDAY 加上 28 天。

(2) 调用 PostgreSQL 的 TO_CHAR 函数或者 MySQL 的 MONTH 函数，CASE 表达式从上述 FIRST_MONDAY + 28 的计算结果里得到与之对应的月份

(3) 第 2 步的计算结果会与内嵌视图返回的 MTH 值相比较。MTH 的值是 CURRENT_DATE 对应的月份。如果两个月份相同，那么这个月就大到需要再加上 28 天，因而 CASE 表达式会返回 FIRST_MONDAY + 28。如果两个月份不同，那么加上 28 天就会超出当前月份的范围，因而 CASE 表达式会返回 FIRST_MONDAY + 21。显而易见，要么加上 28 天，要么加上 21 天，我们只需要考虑这两种状况。



我们甚至可以扩展本解决方案，分别加上 7 天和 14 天就能得到当前月份的第 2 个和第 3 个星期一。

9.7 生成日历

1. 问题

你想为当前月份生成一个日历。日历的格式应该像我们桌面上摆放的台历那样横向有 7 列，（通常）纵向有 5 行。

2. 解决方案

下述每种解决方案都略有不同之处，但它们解决问题的思路是相同的。列出当前月份的每一天，然后根据每一天是星期几确定输出顺序以生成日历。

日历有许多种不同的格式。例如，Unix 下的 cal 命令输出的格式是从星期日到星期六。本实例基于 ISO 标准，按照星期一到星期五的顺序生成日历。掌握了本实例提供的解决方案之后，就能根据自己的喜好轻松地重新安排日历的格式了。



如果我们试图使用 SQL 完成各种格式化操作以便让输出结果更具可读性的话，那么查询语句也会变得更长。不要被这些长长的查询吓住；如果把本实例里的各种查询分解开，并一段一段地执行的话，我们就会发现它们实际上已经足够简洁了。

DB2

使用 WITH 递归查询列出当前月份的每一天，然后使用 CASE 表达式和 MAX 函数根据每一天是星期几编排输出顺序。

```
1   with x(dy,dm,mth,dw,wk)
2     as (
3  select (current_date -day(current_date) day +1 day) dy,
4          day((current_date -day(current_date) day +1 day)) dm,
5          month(current_date) mth,
6          dayofweek(current_date -day(current_date) day +1 day) dw,
7          week_iso(current_date -day(current_date) day +1 day) wk
8    from t1
9   union all
10 select dy+1 day, day(dy+1 day), mth,
11        dayofweek(dy+1 day), week_iso(dy+1 day)
12   from x
13  where month(dy+1 day) = mth
14  )
```

```
15  select max(case dw when 2 then dm end) as Mo,
16         max(case dw when 3 then dm end) as Tu,
17         max(case dw when 4 then dm end) as We,
18         max(case dw when 5 then dm end) as Th,
19         max(case dw when 6 then dm end) as Fr,
20         max(case dw when 7 then dm end) as Sa,
21         max(case dw when 1 then dm end) as Su
22    from x
23   group by wk
24   order by wk
```

Oracle

使用 CONNECT BY 递归查询列出当前月份的每一天，然后使用 CASE 表达式和 MAX 函数根据每一天是星期几编排输出顺序。

```
 1  with x
 2    as (
 3  select *
 4    from (
 5  select to_char(trunc(sysdate,'mm')+level-1,'iw') wk,
 6         to_char(trunc(sysdate,'mm')+level-1,'dd') dm,
 7         to_number(to_char(trunc(sysdate,'mm')+level-1,'d')) dw,
 8         to_char(trunc(sysdate,'mm')+level-1,'mm') curr_mth,
 9         to_char(sysdate,'mm') mth
10    from dual
11   connect by level <= 31
12         )
13   where curr_mth = mth
14  )
15  select max(case dw when 2 then dm end) Mo,
16         max(case dw when 3 then dm end) Tu,
17         max(case dw when 4 then dm end) We,
18         max(case dw when 5 then dm end) Th,
19         max(case dw when 6 then dm end) Fr,
20         max(case dw when 7 then dm end) Sa,
21         max(case dw when 1 then dm end) Su
22    from x
23   group by wk
24   order by wk
```

PostgreSQL

使用 GENERATE_SERIES 函数列出当前月份的每一天。然后使用 CASE 表达式和 MAX 函数根据每一天是星期几编排输出顺序。

```
 1  select max(case dw when 2 then dm end) as Mo,
 2         max(case dw when 3 then dm end) as Tu,
 3         max(case dw when 4 then dm end) as We,
 4         max(case dw when 5 then dm end) as Th,
 5         max(case dw when 6 then dm end) as Fr,
 6         max(case dw when 7 then dm end) as Sa,
 7         max(case dw when 1 then dm end) as Su
 8    from (
 9  select *
10    from (
```

```
11 select cast(date_trunc('month',current_date) as date)+x.id,
12        to_char(
13           cast(
14     date_trunc('month',current_date)
15                as date)+x.id,'iw') as wk,
16        to_char(
17           cast(
18     date_trunc('month',current_date)
19                as date)+x.id,'dd') as dm,
20         cast(
21      to_char(
22         cast(
23   date_trunc('month',current_date)
24                 as date)+x.id,'d') as integer) as dw,
25         to_char(
26            cast(
27      date_trunc('month',current_date)
28                 as date)+x.id,'mm') as curr_mth,
29         to_char(current_date,'mm') as mth
30   from generate_series (0,31) x(id)
31        ) x
32  where mth = curr_mth
33        ) y
34  group by wk
35  order by wk
```

MySQL

使用 T500 列出当前月份的每一天，然后使用 CASE 表达式和 MAX 函数根据每一天是星期几编排输出顺序。

```
 1 select max(case dw when 2 then dm end) as Mo,
 2        max(case dw when 3 then dm end) as Tu,
 3        max(case dw when 4 then dm end) as We,
 4        max(case dw when 5 then dm end) as Th,
 5        max(case dw when 6 then dm end) as Fr,
 6        max(case dw when 7 then dm end) as Sa,
 7        max(case dw when 1 then dm end) as Su
 8   from (
 9 select date_format(dy,'%u') wk,
10        date_format(dy,'%d') dm,
11        date_format(dy,'%w')+1 dw
12   from (
13 select adddate(x.dy,t500.id-1) dy,
14        x.mth
15   from (
16 select adddate(current_date,-dayofmonth(current_date)+1) dy,
17        date_format(
18            adddate(current_date,
19                    -dayofmonth(current_date)+1),
20                    '%m') mth
21   from t1
22        ) x,
23          t500
24  where t500.id <= 31
```

```
25    and date_format(adddate(x.dy,t500.id-1),'%m') = x.mth
26        ) y
27        ) z
28  group by wk
29  order by wk
```

SQL Server

使用 WITH 递归查询列出当前月份的每一天，然后使用 CASE 表达式和 MAX 函数根据每一天是星期几编排输出顺序。

```
 1 with x(dy,dm,mth,dw,wk)
 2   as (
 3 select dy,
 4        day(dy) dm,
 5        datepart(m,dy) mth,
 6        datepart(dw,dy) dw,
 7        case when datepart(dw,dy) = 1
 8             then datepart(ww,dy)-1
 9             else datepart(ww,dy)
10        end wk
11   from (
12 select dateadd(day,-day(getdate())+1,getdate()) dy
13   from t1
14        ) x
15  union all
16 select dateadd(d,1,dy), day(dateadd(d,1,dy)), mth,
17        datepart(dw,dateadd(d,1,dy)),
18        case when datepart(dw,dateadd(d,1,dy)) = 1
19             then datepart(wk,dateadd(d,1,dy))-1
20             else datepart(wk,dateadd(d,1,dy))
21        end
22   from x
23  where datepart(m,dateadd(d,1,dy)) = mth
24 )
25 select max(case dw when 2 then dm end) as Mo,
26        max(case dw when 3 then dm end) as Tu,
27        max(case dw when 4 then dm end) as We,
28        max(case dw when 5 then dm end) as Th,
29        max(case dw when 6 then dm end) as Fr,
30        max(case dw when 7 then dm end) as Sa,
31        max(case dw when 1 then dm end) as Su
32   from x
33  group by wk
34  order by wk
```

3. 讨论

DB2

首先要列出当前月份的每一天。这要用到 WITH 递归查询（如果你使用的 DB2 版本不支持 WITH，则不妨借助类似 T500 这样的数据透视表，具体参考 MySQL 的解决方案）。除了当前月份的每一天（别名 DM），我们还要提取出该日期的不同组成部分：它是星期几（别名 DW），当前的月份（别名 MTH），以及符合 ISO 标准的周序号（别名 WK）。实际的递归操作发生之前（UNION ALL 之前的部分）的递归视图 X 的查询结果如下所示。

```
select (current_date -day(current_date) day +1 day) dy,
       day((current_date -day(current_date) day +1 day)) dm,
       month(current_date) mth,
       dayofweek(current_date -day(current_date) day +1 day) dw,
       week_iso(current_date -day(current_date) day +1 day) wk
  from t1

DY          DM MTH         DW WK
----------- -- --- ---------- --
01-JUN-2005 01  06          4 22
```

接下来就要不断地递增 DM 值（在当前月份里向前推进），直至到达月末。因为我们会遍历当前月份中的每一天，也就能逐一获得每一天是星期几，以及相应的符合 ISO 标准的周序号。部分查询结果显示如下所示。

```
with x(dy,dm,mth,dw,wk)
  as (
select (current_date -day(current_date) day +1 day) dy,
       day((current_date -day(current_date) day +1 day)) dm,
       month(current_date) mth,
       dayofweek(current_date -day(current_date) day +1 day) dw,
       week_iso(current_date -day(current_date) day +1 day) wk
  from t1
 union all
 select dy+1 day, day(dy+1 day), mth,
        dayofweek(dy+1 day), week_iso(dy+1 day)
   from x
  where month(dy+1 day) = mth
)
select *
  from x

DY          DM MTH         DW WK
----------- -- --- ---------- --
01-JUN-2005 01  06          4 22
02-JUN-2005 02  06          5 22
...
21-JUN-2005 21  06          3 25
22-JUN-2005 22  06          4 25
...
30-JUN-2005 30  06          5 26
```

到目前为止，得到的结果包括：当前月份的每一天，两位数字表示的日期，两位数字表示的月份，一位数字表示的星期几（1 ～ 7 分别代表从星期日到星期六的每一天），以及两位数字表示的、符合 ISO 标准的周序号。有了这些信息，我们就能使用 CASE 表达式来决定每一个 DM 值（当前月份的每一天）会落到一周的哪一天。部分查询结果如下所示。

```
with x(dy,dm,mth,dw,wk)
  as (
select (current_date -day(current_date) day +1 day) dy,
       day((current_date -day(current_date) day +1 day)) dm,
       month(current_date) mth,
       dayofweek(current_date -day(current_date) day +1 day) dw,
```

```
        week_iso(current_date -day(current_date) day +1 day) wk
  from t1
 union all
 select dy+1 day, day(dy+1 day), mth,
       dayofweek(dy+1 day), week_iso(dy+1 day)
  from x
 where month(dy+1 day) = mth
)
select wk,
       case dw when 2 then dm end as Mo,
       case dw when 3 then dm end as Tu,
       case dw when 4 then dm end as We,
       case dw when 5 then dm end as Th,
       case dw when 6 then dm end as Fr,
       case dw when 7 then dm end as Sa,
       case dw when 1 then dm end as Su
  from x

WK MO TU WE TH FR SA SU
-- -- -- -- -- -- -- --
22       01
22          02
22             03
22                04
22                   05
23 06
23    07
23       08
23          09
23             10
23                11
23                   12
```

在上面的部分输出结果里，可以看到一周里的每一天都作为单独的一行被返回。现在需要以周为单位为数据分组，并把同一周的七天合并为一行。调用聚合函数 MAX，并按照 WK（符合 ISO 标准的周序号）分组，这样就能把一周七天合并到一行里。为了合理地安排输出格式并保证按日期顺序输出，还要根据 WK 对结果做排序。最终结果如下所示。

```
with x(dy,dm,mth,dw,wk)
  as (
select (current_date -day(current_date) day +1 day) dy,
       day((current_date -day(current_date) day +1 day)) dm,
       month(current_date) mth,
       dayofweek(current_date -day(current_date) day +1 day) dw,
       week_iso(current_date -day(current_date) day +1 day) wk
  from t1
 union all
 select dy+1 day, day(dy+1 day), mth,
       dayofweek(dy+1 day), week_iso(dy+1 day)
  from x
 where month(dy+1 day) = mth
)
select max(case dw when 2 then dm end) as Mo,
       max(case dw when 3 then dm end) as Tu,
```

```
       max(case dw when 4 then dm end) as We,
       max(case dw when 5 then dm end) as Th,
       max(case dw when 6 then dm end) as Fr,
       max(case dw when 7 then dm end) as Sa,
       max(case dw when 1 then dm end) as Su
  from x
 group by wk
 order by wk

MO TU WE TH FR SA SU
-- -- -- -- -- -- --
      01 02 03 04 05
06 07 08 09 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

Oracle

首先使用 CONNECT BY 递归查询生成当前月份的每一天。如果手边没有 Oracle 9*i* 或者更高版本的数据库，我们就无法按这种方式使用 CONNECT BY。如此一来，则需要借助 MySQL 解决方案里出现过的 T500 这样的数据透视表。

除了当前月份的每一天之外，我们还需要其他信息：当前月份每一天的日期部分（别名 DM），每一天分别是星期几（别名 DW），当前的月份（别名 MTH），以及符合 ISO 标准的周序号（别名 WK）。WITH 视图 X 里与当前月份第一天相关的查询结果如下所示。

```
select trunc(sysdate,'mm') dy,
       to_char(trunc(sysdate,'mm'),'dd') dm,
       to_char(sysdate,'mm') mth,
       to_number(to_char(trunc(sysdate,'mm'),'d')) dw,
       to_char(trunc(sysdate,'mm'),'iw') wk
  from dual

DY          DM MT         DW WK
----------- -- -- ---------- --
01-JUN-2005 01 06          4 22
```

接下来就要不断地递增 DM 值（在当前月份里向前推进），直至到达月末。因为我们会遍历当前月份的每一天，也就能逐一获得每一天是星期几，以及相应的符合 ISO 标准的周序号。部分结果如下所示（为了增加可读性，额外添加了每一天的日期）。

```
with x
  as (
select *
  from (
select trunc(sysdate,'mm')+level-1 dy,
       to_char(trunc(sysdate,'mm')+level-1,'iw') wk,
       to_char(trunc(sysdate,'mm')+level-1,'dd') dm,
       to_number(to_char(trunc(sysdate,'mm')+level-1,'d')) dw,
       to_char(trunc(sysdate,'mm')+level-1,'mm') curr_mth,
       to_char(sysdate,'mm') mth
  from dual
 connect by level <= 31
       )
```

```
 where curr_mth = mth
)
select *
  from x

DY          WK DM         DW CU MT
----------- -- -- ---------- -- --
01-JUN-2005 22 01          4 06 06
02-JUN-2005 22 02          5 06 06
...
21-JUN-2005 25 21          3 06 06
22-JUN-2005 25 22          4 06 06
...
30-JUN-2005 26 30          5 06 06
```

到目前为止，当前月份的每一天都作为单独的一行被返回。每一行包括：两位数字表示的日期，两位数字表示的月份，一位数字表示的星期几（1 ～ 7 分别代表从星期日到星期六的每一天），以及两位数字表示的、符合 ISO 标准的周序号。有了这些信息，我们就能使用 CASE 表达式来决定每一个 DM 值（当前月份的每一天）会落到一周的哪一天。部分查询结果如下所示。

```
with x
  as (
select *
  from (
select trunc(sysdate,'mm')+level-1 dy,
       to_char(trunc(sysdate,'mm')+level-1,'iw') wk,
       to_char(trunc(sysdate,'mm')+level-1,'dd') dm,
       to_number(to_char(trunc(sysdate,'mm')+level-1,'d')) dw,
       to_char(trunc(sysdate,'mm')+level-1,'mm') curr_mth,
       to_char(sysdate,'mm') mth
  from dual
 connect by level <= 31
       )
 where curr_mth = mth
)
select wk,
       case dw when 2 then dm end as Mo,
       case dw when 3 then dm end as Tu,
       case dw when 4 then dm end as We,
       case dw when 5 then dm end as Th,
       case dw when 6 then dm end as Fr,
       case dw when 7 then dm end as Sa,
       case dw when 1 then dm end as Su
  from x

WK MO TU WE TH FR SA SU
-- -- -- -- -- -- -- --
22       01
22          02
22             03
22                04
22                   05
23 06
23    07
```

```
23       08
23          09
23             10
23                11
23                   12
```

在上面的部分输出结果里，可以看到一周里的每一天都作为单独的一行被返回，而日期则被放置于 7 列中与 DW 值相对应的那一列。我们需要把一周七天都归并到一行里去。调用聚合函数 MAX，并按照 WK（符合 ISO 标准的周序号）分组，这样就能把一周七天合并到一行了。为了保证按日期顺序输出，还要根据 WK 对结果排序，最终结果如下所示。

```
with x
  as (
select *
  from (
select to_char(trunc(sysdate,'mm')+level-1,'iw') wk,
       to_char(trunc(sysdate,'mm')+level-1,'dd') dm,
       to_number(to_char(trunc(sysdate,'mm')+level-1,'d')) dw,
       to_char(trunc(sysdate,'mm')+level-1,'mm') curr_mth,
       to_char(sysdate,'mm') mth
  from dual
 connect by level <= 31
       )
 where curr_mth = mth
)
select max(case dw when 2 then dm end) Mo,
       max(case dw when 3 then dm end) Tu,
       max(case dw when 4 then dm end) We,
       max(case dw when 5 then dm end) Th,
       max(case dw when 6 then dm end) Fr,
       max(case dw when 7 then dm end) Sa,
       max(case dw when 1 then dm end) Su
  from x
 group by wk
 order by wk

MO TU WE TH FR SA SU
-- -- -- -- -- -- --
      01 02 03 04 05
06 07 08 09 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

PostgreSQL

使用 GENERATE_SERIES 函数把当前月份的每一天都当作单独的一行返回。如果你使用的 PostgreSQL 版本不支持 GENERATE_SERIES，不妨参考 MySQL 解决方案，改为借助一个数据透视表来实现同样的功能。

对于当前月份的每一天，分别提取出下列信息：当前月份每一天的日期部分（别名 DM），每一天分别是星期几（别名 DW），当前的月份（别名 MTH），以及符合 ISO 标准的周序号（别名 WK）。尽管格式化和显式类型转换相关的代码大大降低了本解决方案的可读性，但整个查询其实并不复杂。内嵌视图 X 的部分查询结果如下所示。

```
select cast(date_trunc('month',current_date) as date)+x.id as dy,
       to_char(
          cast(
    date_trunc('month',current_date)
                 as date)+x.id,'iw') as wk,
       to_char(
          cast(
    date_trunc('month',current_date)
                 as date)+x.id,'dd') as dm,
       cast(
    to_char(
       cast(
 date_trunc('month',current_date)
                 as date)+x.id,'d') as integer) as dw,
       to_char(
          cast(
    date_trunc('month',current_date)
                 as date)+x.id,'mm') as curr_mth,
       to_char(current_date,'mm') as mth
  from generate_series (0,31) x(id)

DY          WK DM         DW CU MT
----------- -- -- ---------- -- --
01-JUN-2005 22 01          4 06 06
02-JUN-2005 22 02          5 06 06
...
21-JUN-2005 25 21          3 06 06
22-JUN-2005 25 22          4 06 06
...
30-JUN-2005 26 30          5 06 06
```

注意，当遍历当前月份的每一天时，我们同时能知道每一天是星期几，以及符合 ISO 标准的周序号。为了保证遍历的范围不超出当前月份，我们按照条件 CURR_MTH = MTH 对返回的日期做了过滤（每个日期所对应的月份应该是当前月份）。到目前为止，得到的结果包括：两位数字表示的日期，两位数字表示的月份，一位数字表示的星期几（1 ~ 7 分别代表从星期日到星期六的每一天），以及两位数字表示的、符合 ISO 标准的周序号。下一步需要使用 CASE 表达式来决定每一个 DM 值（当前月份的每一天）会落到一周中的哪一天。部分查询结果如下所示。

```
select case dw when 2 then dm end as Mo,
       case dw when 3 then dm end as Tu,
       case dw when 4 then dm end as We,
       case dw when 5 then dm end as Th,
       case dw when 6 then dm end as Fr,
       case dw when 7 then dm end as Sa,
       case dw when 1 then dm end as Su
  from (
select *
  from (
select cast(date_trunc('month',current_date) as date)+x.id,
       to_char(
          cast(
     date_trunc('month',current_date)
                 as date)+x.id,'iw') as wk,
        to_char(
```

```
            cast(
      date_trunc('month',current_date)
                 as date)+x.id,'dd') as dm,
         cast(
      to_char(
         cast(
   date_trunc('month',current_date)
                 as date)+x.id,'d') as integer) as dw,
         to_char(
            cast(
      date_trunc('month',current_date)
                 as date)+x.id,'mm') as curr_mth,
         to_char(current_date,'mm') as mth
    from generate_series (0,31) x(id)
         ) x
   where mth = curr_mth
         ) y
```

```
WK MO TU WE TH FR SA SU
-- -- -- -- -- -- -- --
22       01
22          02
22             03
22                04
22                   05
23 06
23    07
23       08
23          09
23             10
23                11
23                   12
```

在上面的部分输出结果里，可以看到一周里的每一天都被作为单独的一行被返回，而日期则被放置于 7 列中与 DW 值相对应的那一列。我们需要把一周七天都合并到一行中。因此接下来要调用聚合函数 MAX，并按照 WK（符合 ISO 标准的周序号）分组。这样一来，我们就能把一周七天合并到一行里去了，就像在真实的日历上看到的那样。为了保证按日期顺序输出，还要根据 WK 对结果做排序。最终结果如下所示。

```
select max(case dw when 2 then dm end) as Mo,
       max(case dw when 3 then dm end) as Tu,
       max(case dw when 4 then dm end) as We,
       max(case dw when 5 then dm end) as Th,
       max(case dw when 6 then dm end) as Fr,
       max(case dw when 7 then dm end) as Sa,
       max(case dw when 1 then dm end) as Su
  from (
select *
  from (
select cast(date_trunc('month',current_date) as date)+x.id,
       to_char(
          cast(
    date_trunc('month',current_date)
                as date)+x.id,'iw') as wk,
       to_char(
          cast(
```

```
   date_trunc('month',current_date)
              as date)+x.id,'dd') as dm,
     cast(
   to_char(
     cast(
 date_trunc('month',current_date)
              as date)+x.id,'d') as integer) as dw,
     to_char(
       cast(
   date_trunc('month',current_date)
              as date)+x.id,'mm') as curr_mth,
     to_char(current_date,'mm') as mth
  from generate_series (0,31) x(id)
       ) x
 where mth = curr_mth
       ) y
 group by wk
 order by wk

MO TU WE TH FR SA SU
-- -- -- -- -- -- --
      01 02 03 04 05
06 07 08 09 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

MySQL

首先为当前月份的每一天生成单独的一行。为了实现此目的，需要使用 T500 表。在当前月份第一天的基础上依次加上 T500 表的每一个值，就能得到当前月份的每一天了。

对于每一个日期，需要提取出如下的信息：当前月份每一天的日期部分（别名 DM），每一天分别是星期几（别名 DW），当前的月份（别名 MTH），以及符合 ISO 标准的周序号（别名 WK）。内嵌视图 X 返回当前月份的第一天，以及两位数字表示的当前月份。结果如下所示。

```
select adddate(current_date,-dayofmonth(current_date)+1) dy,
       date_format(
       adddate(current_date,
               -dayofmonth(current_date)+1),
               '%m') mth
  from t1

DY          MT
----------- --
01-JUN-2005 06
```

下一步要遍历当前月份，从第一天开始，依次返回当前月份的每一天。注意，我们会遍历当前月份的每一天，并返回每个日期是星期几以及符合 ISO 标准的周序号。为了保证遍历操作不超出范围，只筛选出那些属于当前月份的日期（每一天对应的月份应该等于当前日期所属的月份）。内嵌视图 Y 的部分查询结果如下所示。

```
select date_format(dy,'%u') wk,
       date_format(dy,'%d') dm,
       date_format(dy,'%w')+1 dw
```

```
  from (
select adddate(x.dy,t500.id-1) dy,
       x.mth
  from (
select adddate(current_date,-dayofmonth(current_date)+1) dy,
       date_format(
           adddate(current_date,
                   -dayofmonth(current_date)+1),
                   '%m') mth
  from t1
       ) x,
         t500
 where t500.id <= 31
   and date_format(adddate(x.dy,t500.id-1),'%m') = x.mth
       ) y

WK DM         DW
-- -- ----------
22 01          4
22 02          5
...
25 21          3
25 22          4
...
26 30          5
```

对于当前月份的每一天，现在知道了下列信息：两位数字表示的日期部分（DM），一位数字表示的星期几（DW），以及两位数字表示的、符合 ISO 标准的周序号（WK）。有了这些信息之后，我们就能借助 CASE 表达式来决定每一个 DM 值（当前月份的每一天）会落到一周的哪一天。部分查询结果如下所示。

```
select case dw when 2 then dm end as Mo,
       case dw when 3 then dm end as Tu,
       case dw when 4 then dm end as We,
       case dw when 5 then dm end as Th,
       case dw when 6 then dm end as Fr,
       case dw when 7 then dm end as Sa,
       case dw when 1 then dm end as Su
  from (
select date_format(dy,'%u') wk,
       date_format(dy,'%d') dm,
       date_format(dy,'%w')+1 dw
  from (
select adddate(x.dy,t500.id-1) dy,
       x.mth
  from (
select adddate(current_date,-dayofmonth(current_date)+1) dy,
       date_format(
           adddate(current_date,
                   -dayofmonth(current_date)+1),
                   '%m') mth
  from t1
       ) x,
         t500
```

```
 where t500.id <= 31
   and date_format(adddate(x.dy,t500.id-1),'%m') = x.mth
       ) y
       ) z

WK MO TU WE TH FR SA SU
-- -- -- -- -- -- -- --
22       01
22          02
22             03
22                04
22                   05
23 06
23    07
23       08
23          09
23             10
23                11
23                   12
```

在上面的部分输出结果里，可以看到一周里的每一天都作为单独的一行被返回。每一行里，日期值都被放置于与 DW 值相对应的那一列。现在我们需要把一周七天都归并到一行里去。因此要调用聚合函数 MAX，并按照 WK（符合 ISO 标准的周序号）分组。为保证按日期顺序输出，还要根据 WK 对结果排序。最终结果如下所示。

```
select max(case dw when 2 then dm end) as Mo,
       max(case dw when 3 then dm end) as Tu,
       max(case dw when 4 then dm end) as We,
       max(case dw when 5 then dm end) as Th,
       max(case dw when 6 then dm end) as Fr,
       max(case dw when 7 then dm end) as Sa,
       max(case dw when 1 then dm end) as Su
  from (
select date_format(dy,'%u') wk,
       date_format(dy,'%d') dm,
       date_format(dy,'%w')+1 dw
  from (
select adddate(x.dy,t500.id-1) dy,
       x.mth
  from (
select adddate(current_date,-dayofmonth(current_date)+1) dy,
       date_format(
           adddate(current_date,
                   -dayofmonth(current_date)+1),
                   '%m') mth
  from t1
       ) x,
         t500
 where t500.id <= 31
   and date_format(adddate(x.dy,t500.id-1),'%m') = x.mth
       ) y
       ) z
 group by wk
 order by wk
```

```
MO TU WE TH FR SA SU
-- -- -- -- -- -- --
      01 02 03 04 05
06 07 08 09 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

SQL Server

首先把当前月份的每一天当作单独的一行返回，可以使用 WITH 递归查询来做到这一点。如果你使用的 SQL Server 版本不支持 WITH 递归查询，可以参考 MySQL 的解决方案，借助数据透视表达到同样的目的。对于每一行返回的值，需要获取如下信息：当前月份每一天的日期部分（别名 DM），每一天分别是星期几（别名 DW），当前的月份（别名 MTH），以及符合 ISO 标准的周序号（别名 WK）。实际的递归操作发生之前（UNION ALL 之前的部分）的递归视图 X 的查询结果如下所示。

```
select dy,
       day(dy) dm,
       datepart(m,dy) mth,
       datepart(dw,dy) dw,
       case when datepart(dw,dy) = 1
            then datepart(ww,dy)-1
            else datepart(ww,dy)
       end wk
  from (
select dateadd(day,-day(getdate())+1,getdate()) dy
  from t1
       )x

DY          DM MTH         DM WK
----------- -- --- ---------- --
01-JUN-2005  1   6          4 23
```

接下来就要不断地递增 DM 值（在当前月份里向前推进），直至到达月末。因为我们会遍历当前月份的每一天，也就能逐一获取到每一天是星期几，以及相应的符合 ISO 标准的周序号。部分查询结果如下所示。

```
  with x(dy,dm,mth,dw,wk)
    as (
select dy,
       day(dy) dm,
       datepart(m,dy) mth,
       datepart(dw,dy) dw,
       case when datepart(dw,dy) = 1
            then datepart(ww,dy)-1
            else datepart(ww,dy)
       end wk
  from (
select dateadd(day,-day(getdate())+1,getdate()) dy
  from t1
       ) x
```

```
 union all
select dateadd(d,1,dy), day(dateadd(d,1,dy)), mth,
       datepart(dw,dateadd(d,1,dy)),
       case when datepart(dw,dateadd(d,1,dy)) = 1
            then datepart(wk,dateadd(d,1,dy))-1
            else datepart(wk,dateadd(d,1,dy))
       end
  from x
 where datepart(m,dateadd(d,1,dy)) = mth
)
select *
  from x

DY          DM MTH         DW WK
----------- -- --- ---------- --
01-JUN-2005 01  06          4 23
02-JUN-2005 02  06          5 23
...
21-JUN-2005 21  06          3 26
22-JUN-2005 22  06          4 26
...
30-JUN-2005 30  06          5 27
```

对于当前月份的每一天，现在得到的结果包括：两位数字表示的日期，两位数字表示的月份，一位数字表示的星期几（1 ～ 7 分别代表从星期日到星期六的每一天），以及两位数字表示的、符合 ISO 标准的周序号。

现在，我们就能使用 CASE 表达式来决定每一个 DM 值（当前月份的每一天）会落到一周的哪一天。部分查询结果如下所示。

```
  with x(dy,dm,mth,dw,wk)
    as (
select dy,
       day(dy) dm,
       datepart(m,dy) mth,
       datepart(dw,dy) dw,
       case when datepart(dw,dy) = 1
            then datepart(ww,dy)-1
            else datepart(ww,dy)
       end wk
  from (
select dateadd(day,-day(getdate())+1,getdate()) dy
  from t1
       ) x
 union all
select dateadd(d,1,dy), day(dateadd(d,1,dy)), mth,
       datepart(dw,dateadd(d,1,dy)),
       case when datepart(dw,dateadd(d,1,dy)) = 1
            then datepart(wk,dateadd(d,1,dy))-1
            else datepart(wk,dateadd(d,1,dy))
       end
  from x
 where datepart(m,dateadd(d,1,dy)) = mth
)
```

```
select case dw when 2 then dm end as Mo,
       case dw when 3 then dm end as Tu,
       case dw when 4 then dm end as We,
       case dw when 5 then dm end as Th,
       case dw when 6 then dm end as Fr,
       case dw when 7 then dm end as Sa,
       case dw when 1 then dm end as Su
  from x

WK MO TU WE TH FR SA SU
-- -- -- -- -- -- -- --
22       01
22          02
22             03
22                04
22                   05
23 06
23    07
23       08
23          09
23             10
23                11
23                   12
```

每一天都作为单独的一行被返回。在每一行里，日期值被放置在与 DW 值相对应的那一列。因此，我们需要把一周七天都归并到一行里去。为达到此目的，针对行数据按照 WK（符合 ISO 标准的周序号）分组，并针对不同的列执行 MAX 函数。查询结果将会以日历的形式输出，如下所示。

```
  with x(dy,dm,mth,dw,wk)
    as (
select dy,
       day(dy) dm,
       datepart(m,dy) mth,
       datepart(dw,dy) dw,
       case when datepart(dw,dy) = 1
            then datepart(ww,dy)-1
            else datepart(ww,dy)
       end wk
  from (
select dateadd(day,-day(getdate())+1,getdate()) dy
  from t1
       ) x
 union all
select dateadd(d,1,dy), day(dateadd(d,1,dy)), mth,
       datepart(dw,dateadd(d,1,dy)),
       case when datepart(dw,dateadd(d,1,dy)) = 1
            then datepart(wk,dateadd(d,1,dy))-1
            else datepart(wk,dateadd(d,1,dy))
       end
  from x
 where datepart(m,dateadd(d,1,dy)) = mth
)
select max(case dw when 2 then dm end) as Mo,
```

```
        max(case dw when 3 then dm end) as Tu,
        max(case dw when 4 then dm end) as We,
        max(case dw when 5 then dm end) as Th,
        max(case dw when 6 then dm end) as Fr,
        max(case dw when 7 then dm end) as Sa,
        max(case dw when 1 then dm end) as Su
  from x
 group by wk
 order by wk

MO TU WE TH FR SA SU
-- -- -- -- -- -- --
      01 02 03 04 05
06 07 08 09 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
```

9.8 列出一年中每个季度的开始日期和结束日期

1. 问题

对于给定年份的四个季度，分别列出它们的开始日期和结束日期。

2. 解决方案

一年有四个季度，因此需要生成 4 行记录。在生成了足够多的行之后，直接调用各个关系数据库管理系统中的日期函数返回每个季度的开始日期和结束日期即可。我们的目标是生成如下所示的结果集，这里以生成当前年份的记录为例。

```
QTR Q_START     Q_END
--- ----------- -----------
  1 01-JAN-2005 31-MAR-2005
  2 01-APR-2005 30-JUN-2005
  3 01-JUL-2005 30-SEP-2005
  4 01-OCT-2005 31-DEC-2005
```

DB2

同时使用 EMP 表和窗口函数 ROW_NUMBER OVER 生成 4 行纪录。除此之外，也可以使用 WITH 子句达到同样目的（正如许多实例的做法一样），甚至也可以借助任何行数不少于 4 行的表。下面的解决方案选择使用 ROW_NUMBER OVER 函数。

```
 1 select quarter(dy-1 day) QTR,
 2        dy-3 month Q_start,
 3        dy-1 day Q_end
 4   from (
 5 select (current_date -
 6          (dayofyear(current_date)-1) day
 7            + (rn*3) month) dy
 8   from (
 9 select row_number()over() rn
10   from emp
11  fetch first 4 rows only
```

```
12         ) x
13         ) y
```

Oracle

使用 ADD_MONTHS 函数找到每个季度的开始日期和结束日期。使用 ROWNUM 代表每个开始日期和结束日期分别属于哪个季度。下面的解决方案借助 EMP 表生成 4 行记录。

```
1 select rownum qtr,
2        add_months(trunc(sysdate,'y'),(rownum-1)*3) q_start,
3        add_months(trunc(sysdate,'y'),rownum*3)-1 q_end
4   from emp
5  where rownum <= 4
```

PostgreSQL

使用 GENERATE_SERIES 函数生成所需的 4 个季度。使用 DATE_TRUNC 函数对每个季度的日期做截断处理，使之仅精确到年份和月份。使用 TO_CHAR 函数计算出每一对开始日期和结束日期分别属于哪个季度。

```
 1 select to_char(dy,'Q') as QTR,
 2        date(
 3          date_trunc('month',dy)-(2*interval '1 month')
 4        ) as Q_start,
 5        dy as Q_end
 6   from (
 7 select date(dy+((rn*3) * interval '1 month'))-1 as dy
 8   from (
 9 select rn, date(date_trunc('year',current_date)) as dy
10   from generate_series(1,4) gs(rn)
11        ) x
12        ) y
```

MySQL

使用 T500 表生成 4 行数据（每个季度一行）。使用 DATE_ADD 和 ADDDATE 函数计算出每个季度的开始日期和结束日期。使用 QUARTER 函数计算出每一对开始和结束日期分别属于哪个季度。

```
 1 select quarter(adddate(dy,-1)) QTR,
 2        date_add(dy,interval -3 month) Q_start,
 3        adddate(dy,-1) Q_end
 4   from (
 5 select date_add(dy,interval (3*id) month) dy
 6   from (
 7 select id,
 8        adddate(current_date,-dayofyear(current_date)+1) dy
 9   from t500
10  where id <= 4
11        ) x
12        ) y
```

SQL Server

使用 WITH 递归查询生成 4 行数据。使用 DATEADD 函数找出开始日期和结束日期。使用 DATEPART 函数计算出每一对开始日期和结束日期分别属于哪个季度。

```
 1  with x (dy,cnt)
 2    as (
 3 select dateadd(d,-(datepart(dy,getdate())-1),getdate()),
 4        1
 5   from t1
 6  union all
 7 select dateadd(m,3,dy), cnt+1
 8   from x
 9  where cnt+1 <= 4
10 )
11 select datepart(q,dateadd(d,-1,dy)) QTR,
12        dateadd(m,-3,dy) Q_start,
13        dateadd(d,-1,dy) Q_end
14   from x
15  order by 1
```

3. 讨论

DB2

首先生成 4 行数据（取值为 1 到 4），每个季度一行。内嵌视图 X 使用窗口函数 ROW_NUMBER OVER 和 FETCH FIRST 子句，仅返回 EMP 表的前 4 行数据，结果如下所示。

```
select row_number()over() rn
  from emp
 fetch first 4 rows only

RN
--
 1
 2
 3
 4
```

然后找出当前年份的第一天，然后加上 *n* 个月，*n* 是 RN 的 3 倍（分别在当前年份第一天的基础上加上 3 个月、6 个月、9 个月和 12 个月），结果如下所示。

```
select (current_date -
         (dayofyear(current_date)-1) day
           + (rn*3) month) dy
  from (
select row_number()over() rn
  from emp
 fetch first 4 rows only
      ) x

DY
-----------
01-APR-2005
01-JUL-2005
01-OCT-2005
01-JAN-2006
```

现在，DY 的值就是每个季度最后一天再加上 1 天的日期。下一步要计算出每个季度的开始日期和结束日期。DY 值减去 1 天就是每个季度的结束日期，DY 减去 3 个月就是每个季度的

开始日期。针对 DY-1 的值（每个季度的最后一天）调用 QUARTER 函数，计算出每一对开始日期和结束日期分别属于哪个季度。

Oracle

把函数 ROWNUM、TRUNC 和 ADD_MONTHS 组合起来使用，就能很容易地解决本问题。为找出每个季度的开始日期，只要在当前年份第一天的基础上加上 *n* 个月即可，*n* 是 (ROWNUM-1)*3（计算结果分别是 0、3、6 和 9）。为找出每个季度的结束日期，在当前年份第一天的基础上分别加上 *n* 个月再减去 1 天，*n* 等于 ROWNUM*3。顺便多说一句，做季度相关的计算时，函数 TO_CHAR 和 TRUNC 搭配上格式化选项 q 非常有用。

PostgreSQL

首先调用 DATE_TRUNC 函数截断当前日期，得到当前年份第一天。然后，加上 *n* 个月再减去 1 天，*n* 是 RN（GENERATE_SERIES 函数的返回值）的 3 倍。结果如下所示。

```
select date(dy+((rn*3) * interval '1 month'))-1 as dy
  from (
select rn, date(date_trunc('year',current_date)) as dy
  from generate_series(1,4) gs(rn)
       ) x

DY
-----------
31-MAR-2005
30-JUN-2005
30-SEP-2005
31-DEC-2005
```

现在得到了每个季度的结束日期，下一步要找出开始日期，用 DY 减去 2 个月，并调用 DATE_TRUNC 函数做截断处理即可。最后，针对每个季度的最后一天（DY）调用 TO_CHAR 函数，计算出每一对开始日期和结束日期分别属于哪个季度。

MySQL

首先调用 ADDDATE 和 DAYOFYEAR 函数找出当前年份的第一天，然后调用 DATE_ADD 函数在当前年份第一天的基础上加上 *n* 个月，*n* 等于 T500.ID 乘以 3。结果如下所示。

```
select date_add(dy,interval (3*id) month) dy
  from (
select id,
       adddate(current_date,-dayofyear(current_date)+1) dy
  from t500
 where id <= 4
       ) x

DY
-----------
01-APR-2005
01-JUL-2005
01-OCT-2005
01-JAN-2006
```

现在得到了每个季度最后一天再加上 1 天的日期；为了找出每个季度的结束日期，只要用

DY 值减去 1 天即可。下一步要找出每个季度的开始日期，从 DY 值里减去 3 个月即可。在每个季度结束日期的基础上调用 QUARTER 函数计算出每一对开始日期和结束日期分别属于哪个季度。

SQL Server

首先找出当前年份的第一天，然后调用 DATEADD 函数逐次加上 *n* 个月，*n* 是当前迭代次数的 3 倍（共有 4 次迭代，因而分别加上了 3×1 个月，3×2 个月，等等）。结果如下所示。

```
with x (dy,cnt)
  as (
select dateadd(d,-(datepart(dy,getdate())-1),getdate()),
       1
  from t1
 union all
select dateadd(m,3,dy), cnt+1
  from x
 where cnt+1 <= 4
)
select dy
  from x

DY
-----------
01-APR-2005
01-JUL-2005
01-OCT-2005
01-JAN-2006
```

DY 值是每个季度最后一天再加上 1 天的日期。为得到每个季度的结束日期，只要调用 DATEADD 函数从 DY 里减去 1 天即可。为找出每个季度的开始日期，调用 DATEADD 函数从 DY 里减去 3 个月。借助 DATEPART 函数根据每个季度的结束日期计算出每一对开始日期和结束日期分别属于哪个季度。

9.9　计算一个季度的开始日期和结束日期

1. 问题

以 *yyyyq* 格式（前面 4 位是年份，最后 1 位是季度序号）给出了年份和季度序号，你希望找出该季度的开始日期和结束日期。

2. 解决方案

本解决方案的关键之处在于如何使用模函数从 *yyyyq* 值里提取出季度序号。（如果不想使用模计算，也可以简单地借助子字符串函数提取出 *yyyyq* 的最后一个数字以得到季度序号，因为我们知道前面 4 位表示年份。）得到了季度序号之后，只要乘以 3，就能计算出该季度最后一个月的月份。下述解决方案用到的内嵌视图 X 里包含了由年份和季度序号组合而成的 4 行数据。查询内嵌视图 X 的话，会得到下面的结果集。

```
select 20051 as yrq from t1 union all
select 20052 as yrq from t1 union all
select 20053 as yrq from t1 union all
```

```
select 20054 as yrq from t1

   YRQ
------
 20051
 20052
 20053
 20054
```

DB2

使用 SUBSTR 函数从内嵌视图 X 里提取出年份，使用 MOD 函数提取出对应的季度序号。

```
 1 select (q_end-2 month) q_start,
 2        (q_end+1 month)-1 day q_end
 3   from (
 4 select date(substr(cast(yrq as char(4)),1,4) ||'-'||
 5        rtrim(cast(mod(yrq,10)*3 as char(2))) ||'-1') q_end
 6   from (
 7 select 20051 yrq from t1 union all
 8 select 20052 yrq from t1 union all
 9 select 20053 yrq from t1 union all
10 select 20054 yrq from t1
11        ) x
12        ) y
```

Oracle

使用 SUBSTR 函数从内嵌视图 X 里提取出年份，使用 MOD 函数提取出对应的季度序号。

```
 1 select add_months(q_end,-2) q_start,
 2        last_day(q_end) q_end
 3   from (
 4 select to_date(substr(yrq,1,4)||mod(yrq,10)*3,'yyyymm') q_end
 5   from (
 6 select 20051 yrq from dual union all
 7 select 20052 yrq from dual union all
 8 select 20053 yrq from dual union all
 9 select 20054 yrq from dual
10        ) x
11        ) y
```

PostgreSQL

使用 SUBSTR 函数从内嵌视图 X 里提取出年份，使用 MOD 函数提取出对应的季度序号。

```
 1 select date(q_end-(2*interval '1 month')) as q_start,
 2        date(q_end+interval '1 month'-interval '1 day') as q_end
 3   from (
 4 select to_date(substr(yrq,1,4)||mod(yrq,10)*3,'yyyymm') as q_end
 5   from (
 6 select 20051 as yrq from t1 union all
 7 select 20052 as yrq from t1 union all
 8 select 20053 as yrq from t1 union all
 9 select 20054 as yrq from t1
10        ) x
11        ) y
```

MySQL

使用 SUBSTR 函数从内嵌视图 X 里提取出年份，使用 MOD 函数提取出对应的季度序号。

```
1 select date_add(
2         adddate(q_end,-day(q_end)+1),
3                 interval -2 month) q_start,
4        q_end
5   from (
6 select last_day(
7     str_to_date(
8          concat(
9          substr(yrq,1,4),mod(yrq,10)*3),'%Y%m')) q_end
10   from (
11 select 20051 as yrq from t1 union all
12 select 20052 as yrq from t1 union all
13 select 20053 as yrq from t1 union all
14 select 20054 as yrq from t1
15        ) x
16        ) y
```

SQL Server

使 SUBSTRING 函数从内嵌视图 X 里提取出年份，使用取模运算符（%）提取出对应的季度序号。

```
1 select dateadd(m,-2,q_end) q_start,
2        dateadd(d,-1,dateadd(m,1,q_end)) q_end
3   from (
4 select cast(substring(cast(yrq as varchar),1,4)+'-'+
5        cast(yrq%10*3 as varchar)+'-1' as datetime) q_end
6   from (
7 select 20051 yrq from t1 union all
8 select 20052 yrq from t1 union all
9 select 20053 yrq from t1 union all
10 select 20054 yrq from t1
11        ) x
12        ) y
```

3. 讨论

DB2

首先找出需要处理的年份和季度序号。调用 SUBSTR 函数从内嵌视图 X（X.YRQ）里提取出年份。为了获取季度序号，用 YRQ 对 10 取模。得到季度序号后，乘以 3 即得到该季度最后一个月的月份，结果如下所示。

```
select substr(cast(yrq as char(4)),1,4) yr,
       mod(yrq,10)*3 mth
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
       ) x
```

```
YR      MTH
---- ------
2005      3
2005      6
2005      9
2005     12
```

现在已经得到了年份和每个季度最后一个月的月份。利用这些值可以构建出每个季度最后一个月第一天的日期。使用连接运算符 || 把年份和月份连接起来，然后使用 DATE 函数将其转换为日期类型。

```
select date(substr(cast(yrq as char(4)),1,4) ||'-'||
       rtrim(cast(mod(yrq,10)*3 as char(2))) ||'-1') q_end
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
       ) x

Q_END
-----------
01-MAR-2005
01-JUN-2005
01-SEP-2005
01-DEC-2005
```

上述 Q_END 值是每个季度最后一个月的第一天。为了计算出该季度的结束日期，只要加上 1 个月，然后再减去 1 天即可。为了找出每个季度的开始日期，要从 Q_END 里减去 2 个月。

Oracle

首先找出需要处理的年份和季度序号。调用 SUBSTR 函数从内嵌视图 X（X.YRQ）里提取出年份。为获取季度序号，用 YRQ 对 10 取模。得到季度序号后，乘以 3 即得到该季度最后一个月的月份，结果如下所示。

```
select substr(yrq,1,4) yr, mod(yrq,10)*3 mth
  from (
select 20051 yrq from dual union all
select 20052 yrq from dual union all
select 20053 yrq from dual union all
select 20054 yrq from dual
       ) x

YR      MTH
---- ------
2005      3
2005      6
2005      9
2005     12
```

现在已经得到了年份和每个季度最后一个月的月份。利用这些值可以构建出每个季度最后一个月第一天的日期。使用连接运算符 || 把年份和月份连接起来，然后使用 TO_DATE 函数将其转换为日期类型。

```
select to_date(substr(yrq,1,4)||mod(yrq,10)*3,'yyyymm') q_end
  from (
select 20051 yrq from dual union all
select 20052 yrq from dual union all
select 20053 yrq from dual union all
select 20054 yrq from dual
       ) x

Q_END
-----------
01-MAR-2005
01-JUN-2005
01-SEP-2005
01-DEC-2005
```

上述 Q_END 值是每个季度最后一个月的第一天。为计算出该季度的结束日期，针对 Q_END 调用 LAST_DAY 函数即可。为找出每个季度的开始日期，要调用 ADD_MONTHS 函数从 Q_END 里减去 2 个月。

PostgreSQL

首先找出需要处理的年份和季度序号。调用 SUBSTR 函数从内嵌视图 X（X.YRQ）里提取出年份。为获取季度序号，用 YRQ 对 10 取模。得到季度序号后，乘以 3 即得到该季度最后一个月的月份，结果如下所示。

```
select substr(yrq,1,4) yr, mod(yrq,10)*3 mth
  from (
select 20051 yrq from dual union all
select 20052 yrq from dual union all
select 20053 yrq from dual union all
select 20054 yrq from dual
       ) x

YR      MTH
---- ------
2005      3
2005      6
2005      9
2005     12
```

现在已经得到了年份和每个季度最后一个月的月份。利用这些值可以构建出每个季度最后一个月第一天的日期。使用连接运算符 || 把年份和月份连接起来，然后使用 TO_DATE 函数将其转换为日期类型。

```
select to_date(substr(yrq,1,4)||mod(yrq,10)*3,'yyyymm') q_end
  from (
select 20051 yrq from dual union all
select 20052 yrq from dual union all
select 20053 yrq from dual union all
select 20054 yrq from dual
       ) x

Q_END
-----------
01-MAR-2005
```

```
01-JUN-2005
01-SEP-2005
01-DEC-2005
```

上述 Q_END 值是每个季度最后一个月的第一天。为计算出该季度的结束日期，在 Q_END 基础上加上 1 个月，再减去 1 天即可。为找出每个季度的开始日期，要从 Q_END 里减去 2 个月。最后要把计算结果转换为日期类型。

MySQL

首先找出需要处理的年份和季度序号。调用 SUBSTR 函数从内嵌视图 X（X.YRQ）里提取出年份。为获取季度序号，用 YRQ 对 10 取模。得到季度序号后，乘以 3 即得到该季度最后一个月的月份，结果如下所示。

```
select substr(yrq,1,4) yr, mod(yrq,10)*3 mth
  from (
select 20051 yrq from dual union all
select 20052 yrq from dual union all
select 20053 yrq from dual union all
select 20054 yrq from dual
       ) x

YR       MTH
---- ------
2005      3
2005      6
2005      9
2005     12
```

现在已经得到了年份和每个季度最后一个月的月份。利用这些值可以构建出每个季度的结束日期。先使用 CONCAT 函数把年份和月份连接起来，然后使用 STR_TO_DATE 函数将其转换为日期类型。最后，调用 LAST_DAY 函数计算出每个季度的最后一天。

```
select last_day(
    str_to_date(
         concat(
         substr(yrq,1,4),mod(yrq,10)*3),'%Y%m')) q_end
  from (
select 20051 as yrq from t1 union all
select 20052 as yrq from t1 union all
select 20053 as yrq from t1 union all
select 20054 as yrq from t1
       ) x

Q_END
-----------
31-MAR-2005
30-JUN-2005
30-SEP-2005
31-DEC-2005
```

因为我们已经知道了每个季度的结束日期，剩下的工作就是要计算出开始日期。调用 DAY 函数计算出每个季度的结束日期分别是当前月份的第几天，接着调用 ADDDATE 函数从 Q_END 里减去该计算结果，这样就得到了前一个月的最后一天；再加上 1 天，就得到了每个

季度最后一个月的第一天。最后，调用 DATE_ADD 函数从上述结果日期里减去 2 个月，至此我们就得到了每个季度的开始日期。

SQL Server

首先找出需要处理的年份和季度序号。调用 SUBSTRING 函数从内嵌视图 X（X.YRQ）里提取出年份。为获取季度序号，用 YRQ 对 10 取模。得到季度序号后，乘以 3 即得到该季度最后一个月的月份，结果如下所示。

```
select substring(yrq,1,4) yr, yrq%10*3 mth
  from (
select 20051 yrq from dual union all
select 20052 yrq from dual union all
select 20053 yrq from dual union all
select 20054 yrq from dual
       ) x

YR       MTH
---- ------
2005      3
2005      6
2005      9
2005     12
```

现在已经得到了年份和每个季度最后一个月的月份。利用这些值可以构建出每个季度最后一个月第一天的日期。使用连接运算符 + 把年份和月份连接起来，然后使用 CAST 函数将其转换为日期类型。

```
select cast(substring(cast(yrq as varchar),1,4)+'-'+
       cast(yrq%10*3 as varchar)+'-1' as datetime) q_end
  from (
select 20051 yrq from t1 union all
select 20052 yrq from t1 union all
select 20053 yrq from t1 union all
select 20054 yrq from t1
       ) x

Q_END
-----------
01-MAR-2005
01-JUN-2005
01-SEP-2005
01-DEC-2005
```

上述 Q_END 值是每个季度最后一个月的第一天。为计算出该季度的结束日期，只要调用 DATEADD 函数加上 1 个月，然后再减去 1 天即可。为找出每个季度的开始日期，需要调用 DATEADD 函数从 Q_END 里减去 2 个月。

9.10　填补缺失的日期

1. 问题

你需要为给定日期区间里的每一天（每一个月、每一周或者每一年）生成一行数据。类似

的行集常用于生成汇总报表。例如，你想计算每个月新入职的员工人数，只要某个年份有新同事入职，则列出该年度内每个月的数字。仔细分析全体员工的入职日期的话，会发现他们的入职日期都介于 1980 年和 1983 年之间。

```
select distinct
       extract(year from hiredate) as year
  from emp

YEAR
-----
 1980
 1981
 1982
 1983
```

你希望获得从 1980 年到 1983 年间每个月新入职的员工人数，期待得到的部分结果集如下所示。

```
MTH          NUM_HIRED
----------- ----------
01-JAN-1981          0
01-FEB-1981          2
01-MAR-1981          0
01-APR-1981          1
01-MAY-1981          1
01-JUN-1981          1
01-JUL-1981          0
01-AUG-1981          0
01-SEP-1981          2
01-OCT-1981          0
01-NOV-1981          1
01-DEC-1981          2
```

2. 解决方案

麻烦之处在于我们希望为每个月都返回一行数据，即使那个月没有新入职的员工（也就是说，某些月份的计数值可能为 0）。因为在 1980 年和 1983 年间并非每个月都有新入职的员工，我们必须自己生成每个月份对应的记录，然后和 EMP 表的 HIREDATE 做外连接（需要对 HIREDATE 做截断处理，使之精确到月，这样才能与我们生成的月份相匹配）。

DB2

使用 WITH 递归查询为每个月生成一行记录（每个月的第一天是从 1980 年 1 月 1 日到 1983 年 12 月 1 日）。准备好了所需日期区间内的全部月份记录之后，和 EMP 表进行外连接，并使用聚合函数 COUNT 计算每个月新入职的员工人数。

```
1    with x (start_date,end_date)
2      as (
3  select (min(hiredate) -
4           dayofyear(min(hiredate)) day +1 day) start_date,
5         (max(hiredate) -
6           dayofyear(max(hiredate)) day +1 day) +1 year end_date
7    from emp
8   union all
```

```
 9 select start_date +1 month, end_date
10   from x
11  where (start_date +1 month) < end_date
12 )
13 select x.start_date mth, count(e.hiredate) num_hired
14   from x left join emp e
15     on (x.start_date = (e.hiredate-(day(hiredate)-1) day))
16  group by x.start_date
17  order by 1
```

Oracle

使用 CONNECT BY 子句生成 1980 年到 1983 年间每个月的记录。然后外连接 EMP 表，并使用聚合函数 COUNT 计算每个月新入职的员工人数。但是，Oracle 8*i* 及更早版本的数据库既不支持 ANSI 标准的外连接，也不能使用 CONNECT BY 作为行生成器。一个简单的变通办法是使用传统的数据透视表（参考 MySQL 的解决方案）。下面的的解决方案使用了 Oracle 的外连接语法。

```
 1   with x
 2     as (
 3 select add_months(start_date,level-1) start_date
 4   from (
 5 select min(trunc(hiredate,'y')) start_date,
 6        add_months(max(trunc(hiredate,'y')),12) end_date
 7   from emp
 8        )
 9  connect by level <= months_between(end_date,start_date)
10 )
11 select x.start_date MTH, count(e.hiredate) num_hired
12   from x, emp e
13  where x.start_date = trunc(e.hiredate(+),'mm')
14  group by x.start_date
15  order by 1
```

接着，下面展示了 ANSI 语法风格的第二个 Oracle 解决方案。

```
 1   with x
 2     as (
 3 select add_months(start_date,level-1) start_date
 4   from (
 5 select min(trunc(hiredate,'y')) start_date,
 6        add_months(max(trunc(hiredate,'y')),12) end_date
 7   from emp
 8        )
 9  connect by level <= months_between(end_date,start_date)
10 )
11 select x.start_date MTH, count(e.hiredate) num_hired
12   from x left join emp e
13     on (x.start_date = trunc(e.hiredate,'mm'))
14  group by x.start_date
15  order by 1
```

PostgreSQL

为了增加代码的可读性，本解决方案使用视图 V，该视图返回从第一个员工入职当年的

1 月 1 日开始，到最近一个新同事入职当年的 12 月 1 日为止有多少个月。调用 GENERATE_SERIES 函数时，把视图 V 的返回值作为第二个参数，这样就能生成适当数目的月份记录（行）了。准备好了所需日期区间内的全部月份记录之后，和 EMP 表进行外连接，并使用聚合函数 COUNT 计算每个月新入职的员工人数。

```
create view v
as
select cast(
        extract(year from age(last_month,first_month))*12-1
          as integer) as mths
  from (
select cast(date_trunc('year',min(hiredate)) as date) as first_month,
       cast(cast(date_trunc('year',max(hiredate))
             as date) + interval '1 year'
             as date) as last_month
  from emp
       ) x
```

```
 1  select y.mth, count(e.hiredate) as num_hired
 2    from (
 3  select cast(e.start_date + (x.id * interval '1 month')
 4           as date) as mth
 5    from generate_series (0,(select mths from v)) x(id),
 6         ( select cast(
 7                   date_trunc('year',min(hiredate))
 8                     as date) as start_date
 9             from emp ) e
10         ) y left join emp e
11      on (y.mth = date_trunc('month',e.hiredate))
12  group by y.mth
13  order by 1
```

MySQL

使用数据透视表 T500 为 1980 年到 1983 年间每一个月份生成一行记录。然后外连接 EMP 表，并使用聚合函数 COUNT 计算每个月新入职的员工人数。

```
 1  select z.mth, count(e.hiredate) num_hired
 2    from (
 3  select date_add(min_hd,interval t500.id-1 month) mth
 4    from (
 5  select min_hd, date_add(max_hd,interval 11 month) max_hd
 6    from (
 7  select adddate(min(hiredate),-dayofyear(min(hiredate))+1) min_hd,
 8         adddate(max(hiredate),-dayofyear(max(hiredate))+1) max_hd
 9    from emp
10         ) x
11         ) y,
12         t500
13   where date_add(min_hd,interval t500.id-1 month) <= max_hd
14         ) z left join emp e
15      on (z.mth = adddate(
16                 date_add(
17                 last_day(e.hiredate),interval -1 month),1))
```

```
18   group by z.mth
19   order by 1
```

SQL Server

使用 WITH 递归查询为每个月生成一行记录（每个月的第一天是从 1980 年 1 月 1 日到 1983 年 12 月 1 日）。准备好了所需日期区间内的全部月份记录之后，和 EMP 表进行外连接，并使用聚合函数 COUNT 计算每个月新入职的员工人数。

```
 1  with x (start_date,end_date)
 2    as (
 3  select (min(hiredate) -
 4           datepart(dy,min(hiredate))+1) start_date,
 5         dateadd(yy,1,
 6          (max(hiredate) -
 7           datepart(dy,max(hiredate))+1)) end_date
 8    from emp
 9   union all
10  select dateadd(mm,1,start_date), end_date
11    from x
12   where dateadd(mm,1,start_date) < end_date
13  )
14  select x.start_date mth, count(e.hiredate) num_hired
15    from x left join emp e
16      on (x.start_date =
17             dateadd(dd,-day(e.hiredate)+1,e.hiredate))
18   group by x.start_date
19   order by 1
```

3. 讨论

DB2

首先生成 1980 年到 1983 年间每一个月份（实际上是每个月第一天的日期）对应的记录行。先针对 HIREDATE 调用 MIN 和 MAX 函数，然后把计算结果分别传递给 DAYOFYEAR 函数。

```
select (min(hiredate) -
         dayofyear(min(hiredate)) day +1 day) start_date,
       (max(hiredate) -
         dayofyear(max(hiredate)) day +1 day) +1 year end_date
  from emp

START_DATE  END_DATE
----------- -----------
01-JAN-1980 01-JAN-1984
```

下一步是不断地在 START_DATE 基础上加上 1 个月，生成所有必要的月份以构造出最终的结果集。上述 END_DATE 值比它实际应有的值多 1 天。不过，这也没有关系。因为我们要不断地在 START_DATE 基础上加上 1 个月，只要在抵达 END_DATE 之前中断递归操作即可。生成的部分月份如下所示。

```
with x (start_date,end_date)
  as (
select (min(hiredate) -
         dayofyear(min(hiredate)) day +1 day) start_date,
```

```
        (max(hiredate) -
          dayofyear(max(hiredate)) day +1 day) +1 year end_date
  from emp
 union all
select start_date +1 month, end_date
  from x
 where (start_date +1 month) < end_date
)
select *
  from x

START_DATE  END_DATE
----------- ----------
01-JAN-1980 01-JAN-1984
01-FEB-1980 01-JAN-1984
01-MAR-1980 01-JAN-1984
...
01-OCT-1983 01-JAN-1984
01-NOV-1983 01-JAN-1984
01-DEC-1983 01-JAN-1984
```

现在已经列出了我们所需的全部月份，接着要外连接到 EMP.HIREDATE。因为每一个 START_DATE 实际上是当前月份的第一天，做外连接时也要把 EMP.HIREDATE 截断变成当前月份的第一天。最后，要针对 EMP.HIREDATE 调用聚合函数 COUNT。

Oracle

首先生成 1980 年到 1983 年间每一个月份的第一天。同时使用 TRUNC 和 ADD_MONTHS 函数，并针对 HIREDATE 分别调用 MIN 和 MAX 函数，这样就能找到两端的月份。

```
select min(trunc(hiredate,'y')) start_date,
       add_months(max(trunc(hiredate,'y')),12) end_date
  from emp

START_DATE  END_DATE
----------- ----------
01-JAN-1980 01-JAN-1984
```

然后，不断地在 START_DATE 基础上加上若干个月以返回最终结果所需的月份。上述 END_DATE 值比它实际应有的值多 1 天。不过，这样也没有关系。因为我们要不断地在 START_DATE 基础上加上若干个月，只要在抵达 END_DATE 之前中断递归操作即可。生成的部分月份如下所示。

```
with x as (
select add_months(start_date,level-1) start_date
  from (
select min(trunc(hiredate,'y')) start_date,
       add_months(max(trunc(hiredate,'y')),12) end_date
  from emp
       )
 connect by level <= months_between(end_date,start_date)
)
select *
  from x
```

```
START_DATE
-----------
01-JAN-1980
01-FEB-1980
01-MAR-1980
...
01-OCT-1983
01-NOV-1983
01-DEC-1983
```

现在已经列出了我们所需的全部月份，接着要外连接到 EMP.HIREDATE。因为每一个 START_DATE 实际上是当前月份的第一天，做外连接时也要把 EMP.HIREDATE 截断变成当前月份的第一天。最后，针对 EMP.HIREDATE 调用聚合函数 COUNT。

PostgreSQL

本解决方案使用 GENERATE_SERIES 函数返回我们所需的月份。如果手边没有支持 GENERATE_SERIES 函数的 PostgreSQL 版本，可以使用 MySQL 解决方案中的数据透视表的做法。首先要理解视图 V。视图 V 会计算出需要生成多少个月份，我们通过找出给定日期区间的边界值来实现这一点。视图 V 里的内嵌视图 X 针对 HIREDATE 调用 MIN 和 MAX 函数以计算出开始日期和结束日期，结果如下所示。

```
select cast(date_trunc('year',min(hiredate)) as date) as first_month,
       cast(cast(date_trunc('year',max(hiredate))
             as date) + interval '1 year'
             as date) as last_month
  from emp

FIRST_MONTH LAST_MONTH
----------- -----------
01-JAN-1980 01-JAN-1984
```

上述 LAST_MONTH 值比它实际应有的值要多 1 天。不过，这样也没有关系。在计算两个日期之间有多少个月时，只要在计算结果的基础上减去 1 即可。下一步要调用 AGE 函数找出两个日期之间相差多少年，然后乘以 12（要记得减去 1）。

```
select cast(
        extract(year from age(last_month,first_month))*12-1
          as integer) as mths
  from (
select cast(date_trunc('year',min(hiredate)) as date) as first_month,
       cast(cast(date_trunc('year',max(hiredate))
             as date) + interval '1 year'
             as date) as last_month
  from emp
       ) x

MTHS
----
47
```

把视图 V 的返回值作为第 2 个参数传递给 GENERATE_SERIES 函数，这样就能得到所需数目

的月份。下一步是找出开始日期。我们不断在开始日期的基础上加上若干个月以生成所需的月份区间。内嵌视图 Y 针对 MIN(HIREDATE) 调用 DATE_TRUNC 函数以找出开始日期，并利用 GENERATE_SERIES 函数的返回值逐次为该开始日期加上若干个月。部分结果如下所示。

```
select cast(e.start_date + (x.id * interval '1 month')
         as date) as mth
  from generate_series (0,(select mths from v)) x(id),
       ( select cast(
                date_trunc('year',min(hiredate))
                  as date) as start_date
           from emp
        ) e

MTH
-----------
01-JAN-1980
01-FEB-1980
01-MAR-1980
...
01-OCT-1983
01-NOV-1983
01-DEC-1983
```

现在得到了最终结果集所需的每一个月份，接着要外连接到 EMP.HIREDATE，并调用聚合函数 COUNT 计算每个月新入职员工的人数。

MySQL

首先，使用聚合函数 MIN 和 MAX 以及函数 DAYOFYEAR 和 ADDDATE 找出日期区间的边界值。内嵌视图 X 的查询结果如下所示。

```
select adddate(min(hiredate),-dayofyear(min(hiredate))+1) min_hd,
       adddate(max(hiredate),-dayofyear(max(hiredate))+1) max_hd
  from emp

MIN_HD      MAX_HD
----------- -----------
01-JAN-1980 01-JAN-1983
```

下一步，对 MAX_HD 做加法以计算出当前年份的最后一个月。

```
select min_hd, date_add(max_hd,interval 11 month) max_hd
  from (
select adddate(min(hiredate),-dayofyear(min(hiredate))+1) min_hd,
       adddate(max(hiredate),-dayofyear(max(hiredate))+1) max_hd
  from emp
       ) x

MIN_HD      MAX_HD
----------- -----------
01-JAN-1980 01-DEC-1983
```

现在我们知道了日期边界值，接着使用数据透视表 T500 在 MIN_HD 基础上逐一加上若干个月，直到抵达 MAX_HD 值，这样就生成了我们所需要的行记录。部分结果如下所示。

```
select date_add(min_hd,interval t500.id-1 month) mth
  from (
select min_hd, date_add(max_hd,interval 11 month) max_hd
  from (
select adddate(min(hiredate),-dayofyear(min(hiredate))+1) min_hd,
       adddate(max(hiredate),-dayofyear(max(hiredate))+1) max_hd
  from emp
       ) x
       ) y,
       t500
 where date_add(min_hd,interval t500.id-1 month) <= max_hd

MTH
-----------
01-JAN-1980
01-FEB-1980
01-MAR-1980
...
01-OCT-1983
01-NOV-1983
01-DEC-1983
```

现在已经准备好最终结果所需的全部月份，接着外连接到 EMP.HIREDATE（要记得截断 EMP.HIREDATE 值，使之变成当前月份的第一天），并针对 EMP.HIREDATE 调用聚合函数 COUNT 以计算每个月新入职员工的人数。

SQL Server

首先为从 1980 年到 1983 年间每个月份（实际上是每个月的第一天）生成一行记录。然后，针对 HIREDATE 分别执行 MIN 和 MAX 函数，再调用 DAYOFYEAR 函数，这样就能计算出日期区间两端的月份。

```
select (min(hiredate) -
         datepart(dy,min(hiredate))+1) start_date,
       dateadd(yy,1,
        (max(hiredate) -
         datepart(dy,max(hiredate))+1)) end_date
  from emp

START_DATE  END_DATE
----------- -----------
01-JAN-1980 01-JAN-1984
```

下一步要不断地在 START_DATE 基础上加上若干个月以返回最终结果集所需的月份。上述 END_DATE 值比它实际应有的值多 1 天；不过没有关系，因为我们要不断地在 START_DATE 基础上加上若干个月，只要在抵达 END_DATE 之前中断递归操作即可。生成的部分月份如下所示。

```
with x (start_date,end_date)
  as (
select (min(hiredate) -
         datepart(dy,min(hiredate))+1) start_date,
       dateadd(yy,1,
        (max(hiredate) -
```

```
         datepart(dy,max(hiredate))+1)) end_date
  from emp
 union all
select dateadd(mm,1,start_date), end_date
  from x
 where dateadd(mm,1,start_date) < end_date
)
select *
  from x

START_DATE  END_DATE
----------- ----------
01-JAN-1980 01-JAN-1984
01-FEB-1980 01-JAN-1984
01-MAR-1980 01-JAN-1984
...
01-OCT-1983 01-JAN-1984
01-NOV-1983 01-JAN-1984
01-DEC-1983 01-JAN-1984
```

现在已经列出了我们所需的全部月份，接着要外连接到 EMP.HIREDATE。因为每一个 START_DATE 实际上是当前月份的第一天，做外连接时也要把 EMP.HIREDATE 截断变成当前月份的第一天。最后，针对 EMP.HIREDATE 调用聚合函数 COUNT。

9.11 依据特定时间单位检索数据

1. 问题

你想依据指定的月份、星期或者其他时间单位来筛选记录行。例如，你希望找出入职月份是 February（2 月）或者 December（12 月），并且入职当天是 Tuesday（星期二）的所有员工。

2. 解决方案

使用关系数据库管理系统中的函数来找出一个日期值对应的月份和星期。在很多情况下，本实例可能都会有用。试想，如果我们希望依据 HIREDATE 做一些检索，但又想忽略年份，只按照月份检索（或者按照 HIREDATE 里其他我们感兴趣的时间单位来检索），这个时候本实例提供的技巧就有了用武之地。下面给出的解决方案以月份和星期的检索为例。掌握了各种数据库提供的日期格式化函数之后，我们就能很方便地修改这些解决方案以实现按照年份检索，按照季度检索，按照年份和季度的组合检索，按照月份和年份的组合检索，等等。

DB2 和 MySQL

使用函数 MONTHNAME 和 DAYNAME 分别找出员工入职日期所对应的月份和星期。

```
1 select ename
2   from emp
3  where monthname(hiredate) in ('February','December')
4     or dayname(hiredate) = 'Tuesday'
```

Oracle 和 PostgreSQL

使用 TO_CHAR 函数找出员工入职日期对应的月份和星期，使用 RTRIM 函数过滤掉尾部的空格字符。

```
1 select ename
2   from emp
3  where rtrim(to_char(hiredate,'month')) in ('february','december')
4     or rtrim(to_char(hiredate,'day')) = 'tuesday'
```

SQL Server

使用 DATENAME 函数找出员工入职日期对应的月份和星期。

```
1  select ename
2    from emp
3   where datename(m,hiredate) in ('February','December')
4      or datename(dw,hiredate) = 'Tuesday'
```

3. 讨论

对于以上所有解决方案而言，关键在于知道要调用哪个函数以及如何调用。如果想确认每个函数返回的结果值，不妨在 SELECT 子句后面直接调用相应的函数并检查其输出结果。下面列出了 DEPTNO 等于 10 的员工对应的查询结果集。

```
select ename,datename(m,hiredate) mth,datename(dw,hiredate) dw
  from emp
 where deptno = 10

ENAME  MTH       DW
------ --------- ---------
CLARK  June      Tuesday
KING   November  Tuesday
MILLER January   Saturday
```

一旦知道了相关的函数返回的结果值，那么利用上述解决方案里用到的函数来筛选行记录就非常容易了。

9.12　比较特定的日期要素

1. 问题

你想找出哪些员工在同一个月份和同一个工作日入职。例如，一个员工的入职日期是 1988 年 3 月 10 日，星期一；另一个则是 2001 年 3 月 2 日，星期一。那么，由于二者的月份名称和星期值一致，你认为他们两个的入职日期是相匹配的。对于 EMP 表而言，只有 3 个员工符合这种条件。你希望得到的结果集如下所示。

```
MSG
-------------------------------------------------------
JAMES was hired on the same month and weekday as FORD
SCOTT was hired on the same month and weekday as JAMES
SCOTT was hired on the same month and weekday as FORD
```

2. 解决方案

我们希望把一个员工的 HIREDATE 与另一个员工的 HIREDATE 相比较，因而需要对 EMP 表做自连接查询。这样每一种可能的 HIREDATE 组合都会被拿出来比较一下。然后，只需提取出每一个 HIREDATE 的星期值和月份并做比较即可。

DB2

EMP 表自连接之后，使用 DAYOFWEEK 函数返回一个数值表示星期几。使用 MONTHNAME 函数返回月份名称。

```
1 select a.ename ||
2        ' was hired on the same month and weekday as '||
3        b.ename msg
4   from emp a, emp b
5  where (dayofweek(a.hiredate),monthname(a.hiredate)) =
6        (dayofweek(b.hiredate),monthname(b.hiredate))
7    and a.empno < b.empno
8  order by a.ename
```

Oracle 和 PostgreSQL

EMP 自连接之后，使用 TO_CHAR 函数格式化 HIREDATE 得到筛选条件里的星期值和月份。

```
1  select a.ename ||
2         ' was hired on the same month and weekday as '||
3         b.ename as msg
4    from emp a, emp b
5   where to_char(a.hiredate,'DMON') =
6         to_char(b.hiredate,'DMON')
7     and a.empno < b.empno
8   order by a.ename
```

MySQL

EMP 表自连接之后，使用 DATE_FORMAT 函数格式化 HIREDATE 得到筛选条件里的星期值和月份。

```
1  select concat(a.ename,
2         ' was hired on the same month and weekday as ',
3         b.ename) msg
4    from emp a, emp b
5   where date_format(a.hiredate,'%w%M') =
6         date_format(b.hiredate,'%w%M')
7     and a.empno < b.empno
8   order by a.ename
```

SQL Server

EMP 自连接之后，使用 DATENAME 函数格式化 HIREDATE 得到筛选条件里的星期值和月份。

```
1 select a.ename +
2        ' was hired on the same month and weekday as '+
3        b.ename msg
4   from emp a, emp b
5  where datename(dw,a.hiredate) = datename(dw,b.hiredate)
6    and datename(m,a.hiredate)  = datename(m,b.hiredate)
7    and a.empno < b.empno
8  order by a.ename
```

3. 讨论

以上几种解决方案只在格式化 HIREDATE 时用到了不同的日期函数。下面的讨论部分将以 Oracle 和 PostgreSQL 的解决方案为例（因为该解决方案最为简短）；尽管如此，下面的讲解内容也同样适用于其他解决方案。

首先用 EMP 表做自连接查询，这样每个员工都能访问其他员工的 HIREDATE。我们来看一下下面的查询结果（只筛选出了与员工 SCOTT 相关的数据）。

```
select a.ename as scott, a.hiredate as scott_hd,
       b.ename as other_emps, b.hiredate as other_hds
  from emp a, emp b
 where a.ename  = 'SCOTT'
   and a.empno != b.empno

SCOTT      SCOTT_HD    OTHER_EMPS OTHER_HDS
---------- ----------- ---------- -----------
SCOTT      09-DEC-1982 SMITH      17-DEC-1980
SCOTT      09-DEC-1982 ALLEN      20-FEB-1981
SCOTT      09-DEC-1982 WARD       22-FEB-1981
SCOTT      09-DEC-1982 JONES      02-APR-1981
SCOTT      09-DEC-1982 MARTIN     28-SEP-1981
SCOTT      09-DEC-1982 BLAKE      01-MAY-1981
SCOTT      09-DEC-1982 CLARK      09-JUN-1981
SCOTT      09-DEC-1982 KING       17-NOV-1981
SCOTT      09-DEC-1982 TURNER     08-SEP-1981
SCOTT      09-DEC-1982 ADAMS      12-JAN-1983
SCOTT      09-DEC-1982 JAMES      03-DEC-1981
SCOTT      09-DEC-1982 FORD       03-DEC-1981
SCOTT      09-DEC-1982 MILLER     23-JAN-1982
```

通过 EMP 表自连接查询，我们把 SCOTT 的 HIREDATE 和所有其他员工的 HIREDATE 做了比较。筛选条件里的 EMPNO 表明 SCOTT 的 HIREDATE 不会作为 OTHER_HDS 列返回。下一步要使用数据库内置的日期格式化函数比较 HIREDATE 对应的星期值和月份，并只保留相匹配的行。

```
select a.ename as emp1, a.hiredate as emp1_hd,
       b.ename as emp2, b.hiredate as emp2_hd
  from emp a, emp b
 where to_char(a.hiredate,'DMON') =
       to_char(b.hiredate,'DMON')
   and a.empno != b.empno
 order by 1

EMP1       EMP1_HD     EMP2       EMP2_HD
---------- ----------- ---------- -----------
FORD       03-DEC-1981 SCOTT      09-DEC-1982
FORD       03-DEC-1981 JAMES      03-DEC-1981
JAMES      03-DEC-1981 SCOTT      09-DEC-1982
JAMES      03-DEC-1981 FORD       03-DEC-1981
SCOTT      09-DEC-1982 JAMES      03-DEC-1981
SCOTT      09-DEC-1982 FORD       03-DEC-1981
```

现在，HIREDATE 都已经正确地匹配出来了，但是我们得到了 6 行查询结果，而不是前面讲

过的 3 行。这是因为筛选条件 EMPNO 导致了多余的行。使用“不等于”作为筛选条件，我们就没有办法过滤掉反向的查询结果。例如，上述第一行是 FORD 匹配 SCOTT 的结果，而最后一行则是 SCOTT 匹配 FORD。结果集里出现了 6 行数据，从技术上讲虽然没错，实际上却出现了重复数据。我们可以改用“小于”作为筛选条件以避免出现重复数据。（下面的查询语句去掉了 HIREDATE，这样查询结果会更接近最终结果集。）

```
select a.ename as emp1, b.ename as emp2
  from emp a, emp b
 where to_char(a.hiredate,'DMON') =
       to_char(b.hiredate,'DMON')
   and a.empno < b.empno
 order by 1

EMP1       EMP2
---------- ----------
JAMES      FORD
SCOTT      JAMES
SCOTT      FORD
```

最后，只要把上述查询结果连接起来变成符合要求的内容即可。

9.13 识别重叠的日期区间

1. 问题

如果一个员工手头现有的项目尚未结束，却又开始了另一个新项目，那么你希望筛选出相关的数据。我们先看一下 EMP_PROJECT 表的数据。

```
select *
  from emp_project

EMPNO ENAME      PROJ_ID PROJ_START  PROJ_END
----- ---------- ------- ----------- -----------
7782  CLARK            1 16-JUN-2005 18-JUN-2005
7782  CLARK            4 19-JUN-2005 24-JUN-2005
7782  CLARK            7 22-JUN-2005 25-JUN-2005
7782  CLARK           10 25-JUN-2005 28-JUN-2005
7782  CLARK           13 28-JUN-2005 02-JUL-2005
7839  KING             2 17-JUN-2005 21-JUN-2005
7839  KING             8 23-JUN-2005 25-JUN-2005
7839  KING            14 29-JUN-2005 30-JUN-2005
7839  KING            11 26-JUN-2005 27-JUN-2005
7839  KING             5 20-JUN-2005 24-JUN-2005
7934  MILLER           3 18-JUN-2005 22-JUN-2005
7934  MILLER          12 27-JUN-2005 28-JUN-2005
7934  MILLER          15 30-JUN-2005 03-JUL-2005
7934  MILLER           9 24-JUN-2005 27-JUN-2005
7934  MILLER           6 21-JUN-2005 23-JUN-2005
```

看一下上述查询结果，员工 KING 在 PROJ_ID 5 结束之前就开始了 PROJ_ID 8，并且它开始 PROJ_ID 5 的时候，PROJ_ID 2 还没有完结。因此，你希望得到的结果集如下所示。

```
EMPNO ENAME      MSG
----- ---------- ------------------------------
 7782 CLARK      project 7 overlaps project 4
 7782 CLARK      project 10 overlaps project 7
 7782 CLARK      project 13 overlaps project 10
 7839 KING       project 8 overlaps project 5
 7839 KING       project 5 overlaps project 2
 7934 MILLER     project 12 overlaps project 9
 7934 MILLER     project 6 overlaps project 3
```

2. 解决方案

这里的关键之处在于要找出那些 PROJ_START（新项目开始的日期）等于或者大于另一个项目 PROJ_START 的行，以及等于或者小于其他项目 PROJ_END 的行。因此，首先要逐个地比较（同一个员工的）每一个项目和其他项目。通过自连接 EMP_PROJECT 表，我们为每一个员工生成全部可能的项目组合。为找到日期重叠的项目，只要在同一个员工的项目里找出 PROJ_START 介于另一个 PROJ_START 和 PROJ_END 之间的那些行即可。

DB2、PostgreSQL 和 Oracle

EMP_PROJECT 表自连接，然后使用连接运算符 || 为时间上发生了重叠的项目构造出符合要求的输出结果。

```
1  select a.empno,a.ename,
2         'project '||b.proj_id||
3          ' overlaps project '||a.proj_id as msg
4    from emp_project a,
5         emp_project b
6   where a.empno = b.empno
7     and b.proj_start >= a.proj_start
8     and b.proj_start <= a.proj_end
9     and a.proj_id != b.proj_id
```

MySQL

EMP_PROJECT 表自连接，然后使用 CONCAT 函数为时间上发生了重叠的项目构造出符合要求的输出结果。

```
1  select a.empno,a.ename,
2         concat('project ',b.proj_id,
3          ' overlaps project ',a.proj_id) as msg
4    from emp_project a,
5         emp_project b
6   where a.empno = b.empno
7     and b.proj_start >= a.proj_start
8     and b.proj_start <= a.proj_end
9     and a.proj_id != b.proj_id
```

SQL Server

EMP_PROJECT 表自连接，然后使用连接运算符 + 为时间上发生了重叠的项目构造出符合要求的输出结果。

```
1  select a.empno,a.ename,
2         'project '+b.proj_id+
3          ' overlaps project '+a.proj_id as msg
```

```
4    from emp_project a,
          emp_project b
6   where a.empno = b.empno
7     and b.proj_start >= a.proj_start
8     and b.proj_start <= a.proj_end
9     and a.proj_id != b.proj_id
```

3. 讨论

上述各个解决方案之间的差别仅在于字符串连接方式不同，接下来的讨论里会使用 DB2 语法，但仍然能兼顾全部 3 种解决方案。首先自连接 EMP_PROJECT 表，这样 PROJ_START 日期值就能和其他项目逐一地做比较了。下面是针对员工 KING 的自连接查询结果。可以发现，每一个项目是怎样"看见"其他项目的。

```
select a.ename,
       a.proj_id as a_id,
       a.proj_start as a_start,
       a.proj_end as a_end,
       b.proj_id as b_id,
       b.proj_start as b_start
  from emp_project a,
       emp_project b
 where a.ename    = 'KING'
   and a.empno    = b.empno
   and a.proj_id != b.proj_id
  order by 2


ENAME   A_ID A_START     A_END        B_ID B_START
------ ----- ----------- ----------- ----- -----------
KING       2 17-JUN-2005 21-JUN-2005     8 23-JUN-2005
KING       2 17-JUN-2005 21-JUN-2005    14 29-JUN-2005
KING       2 17-JUN-2005 21-JUN-2005    11 26-JUN-2005
KING       2 17-JUN-2005 21-JUN-2005     5 20-JUN-2005
KING       5 20-JUN-2005 24-JUN-2005     2 17-JUN-2005
KING       5 20-JUN-2005 24-JUN-2005     8 23-JUN-2005
KING       5 20-JUN-2005 24-JUN-2005    11 26-JUN-2005
KING       5 20-JUN-2005 24-JUN-2005    14 29-JUN-2005
KING       8 23-JUN-2005 25-JUN-2005     2 17-JUN-2005
KING       8 23-JUN-2005 25-JUN-2005    14 29-JUN-2005
KING       8 23-JUN-2005 25-JUN-2005     5 20-JUN-2005
KING       8 23-JUN-2005 25-JUN-2005    11 26-JUN-2005
KING      11 26-JUN-2005 27-JUN-2005     2 17-JUN-2005
KING      11 26-JUN-2005 27-JUN-2005     8 23-JUN-2005
KING      11 26-JUN-2005 27-JUN-2005    14 29-JUN-2005
KING      11 26-JUN-2005 27-JUN-2005     5 20-JUN-2005
KING      14 29-JUN-2005 30-JUN-2005     2 17-JUN-2005
KING      14 29-JUN-2005 30-JUN-2005     8 23-JUN-2005
KING      14 29-JUN-2005 30-JUN-2005     5 20-JUN-2005
KING      14 29-JUN-2005 30-JUN-2005    11 26-JUN-2005
```

从上面的结果集里可以看到，自连接查询使得寻找重叠日期区间的工作变得容易多了。只要筛选出 B_START 介于 A_START 和 A_END 之间的那些行即可。DB2 解决方案第 7 行和第 8 行的 WHERE 条件做的就是这件事。

```
and b.proj_start >= a.proj_start
and b.proj_start <= a.proj_end
```

找到了所需的行之后，接下来只要连接字符串构造出适当格式的输出结果即可。

如果每一个员工的最大项目个数是固定的，那么 Oracle 用户就可以利用窗口函数 LEAD OVER 来避免使用自连接查询。在某些特定情况下使用自连接查询可能显得代价过于沉重（如果自连接耗费的机器资源远大于 LEAD OVER），这个时候 LEAD OVER 函数就不失为一种简便的方法。例如，考虑下面针对员工 KING 的 LEAD OVER 替代方案。

```
select empno,
       ename,
       proj_id,
       proj_start,
       proj_end,
       case
       when lead(proj_start,1)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,2)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,3)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,4)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       end is_overlap
  from emp_project
 where ename = 'KING'


EMPNO ENAME  PROJ_ID PROJ_START  PROJ_END    IS_OVERLAP
----- ------ ------- ----------- ----------- ----------
 7839 KING         2 17-JUN-2005 21-JUN-2005          5
 7839 KING         5 20-JUN-2005 24-JUN-2005          8
 7839 KING         8 23-JUN-2005 25-JUN-2005
 7839 KING        11 26-JUN-2005 27-JUN-2005
 7839 KING        14 29-JUN-2005 30-JUN-2005
```

对于员工 KING，因为项目个数被限定为 5 个，我们就能使用 LEAD OVER 函数逐一检查全部项目的日期值，而无须自连接查询。现在，离最终的结果集只有一步之遥了。接下来只要把 IS_OVERLAP 不是 Null 的行筛选出来即可。

```
select empno,ename,
       'project '||is_overlap||
        ' overlaps project '||proj_id msg
  from (
select empno,
       ename,
       proj_id,
       proj_start,
       proj_end,
```

```
       case
       when lead(proj_start,1)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,2)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,3)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       when lead(proj_start,4)over(order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(order by proj_start)
       end is_overlap
  from emp_project
 where ename = 'KING'
       )
 where is_overlap is not null


EMPNO ENAME  MSG
----- ------ --------------------------------
 7839 KING   project 5 overlaps project 2
 7839 KING   project 8 overlaps project 5
```

为了让以上解决方案适用于全体员工（而不局限于 KING），要在 LEAD OVER 函数调用里加上针对 ENAME 的分区。

```
select empno,ename,
       'project '||is_overlap||
        ' overlaps project '||proj_id msg
  from (
select empno,
       ename,
       proj_id,
       proj_start,
       proj_end,
       case
       when lead(proj_start,1)over(partition by ename
                                   order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(partition by ename
                              order by proj_start)
       when lead(proj_start,2)over(partition by ename
                                   order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(partition by ename
                              order by proj_start)
       when lead(proj_start,3)over(partition by ename
                                   order by proj_start)
            between proj_start and proj_end
       then lead(proj_id)over(partition by ename
                              order by proj_start)
       when lead(proj_start,4)over(partition by ename
                                   order by proj_start)
```

```
            between proj_start and proj_end
       then lead(proj_id)over(partition by ename
                              order by proj_start)
       end is_overlap
  from emp_project
       )
 where is_overlap is not null

EMPNO ENAME  MSG
----- ------ -------------------------------
 7782 CLARK  project 7 overlaps project 4
 7782 CLARK  project 10 overlaps project 7
 7782 CLARK  project 13 overlaps project 10
 7839 KING   project 5 overlaps project 2
 7839 KING   project 8 overlaps project 5
 7934 MILLER project 6 overlaps project 3
 7934 MILLER project 12 overlaps project 9
```

第10章

区间查询

本章将介绍与区间相关的日常查询。区间在日常生活中很常见。例如，我们每天为之努力工作的项目就限定在一个连续的时间区间里。在 SQL 中，经常需要针对一个区间做检索，或者生成某个值区间，抑或对某个区间内的数据做一些处理。本章的查询将会比前面几章的更复杂一些，但它们也是常用的查询。一旦我们真正掌握了相关技术，将会真切感受到 SQL 的威力。

10.1 定位连续的值区间

1. 问题

你想确定哪些行代表了一系列在时间上连续的项目。考虑下述视图 V 的结果集，它包含了项目编号以及各自的起止日期。

```
select *
  from V

PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2005 02-JAN-2005
      2 02-JAN-2005 03-JAN-2005
      3 03-JAN-2005 04-JAN-2005
      4 04-JAN-2005 05-JAN-2005
      5 06-JAN-2005 07-JAN-2005
      6 16-JAN-2005 17-JAN-2005
      7 17-JAN-2005 18-JAN-2005
      8 18-JAN-2005 19-JAN-2005
      9 19-JAN-2005 20-JAN-2005
     10 21-JAN-2005 22-JAN-2005
     11 26-JAN-2005 27-JAN-2005
```

```
     12 27-JAN-2005 28-JAN-2005
     13 28-JAN-2005 29-JAN-2005
     14 29-JAN-2005 30-JAN-2005
```

除了第一行，其他每一行的 PROJ_START 应该等于前一行的 PROJ_END（“前一行”的定义是其 PROJ_ID 等于当前行的 PROJ_ID 减 1）。仔细查看视图 V 的前 5 行，PROJ_ID 分别等于 1 到 3 的行属于同一“组”，因为每一行的 PROJ_END 都等于后一行的 PROJ_START。我们希望找出一系列连续项目的日期区间，因此希望返回满足“当前行的 PROJ_END 等于下一行的 PROJ_START”这一条件的所有行。如果整个结果集只包含前 5 行，那么我们希望返回的只是最前面的 3 行。（对于视图 V 全部 14 行数据而言）最终的结果集应该如下所示。

```
PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2005 02-JAN-2005
      2 02-JAN-2005 03-JAN-2005
      3 03-JAN-2005 04-JAN-2005
      6 16-JAN-2005 17-JAN-2005
      7 17-JAN-2005 18-JAN-2005
      8 18-JAN-2005 19-JAN-2005
     11 26-JAN-2005 27-JAN-2005
     12 27-JAN-2005 28-JAN-2005
     13 28-JAN-2005 29-JAN-2005
```

我们从结果集中剔除掉了 PROJ_ID 为 4、5、9、10 和 14 的行，因为这些行的 PROJ_END 不等于下一行的 PROJ_START。

2. 解决方案

DB2、MySQL、PostgreSQL 和 SQL Server

使用自连接找出包含连续值的行。

```
1 select v1.proj_id,
2        v1.proj_start,
3        v1.proj_end
4   from V v1, V v2
5  where v1.proj_end = v2.proj_start
```

Oracle

上述解决方案也适用于 Oracle。除此之外，Oracle 还有另一个解决方案，即利用窗口函数 LEAD OVER 去查看“下一行”的 BEGIN_DATE，这样就不必自连接了。

```
1 select proj_id,proj_start,proj_end
2   from (
3 select proj_id,proj_start,proj_end,
4        lead(proj_start)over(order by proj_id) next_proj_start
5   from V
6        )
7 where next_proj_start = proj_end
```

3. 讨论

DB2、MySQL、PostgreSQL 和 SQL Server

通过视图 V 的自连接，每一行都可以与其他行进行比较。考虑查询 PROJ_ID 为 1 和 4 时得

到的部分结果集。

```
select v1.proj_id as v1_id,
       v1.proj_end as v1_end,
       v2.proj_start as v2_begin,
       v2.proj_id as v2_id
  from v v1, v v2
 where v1.proj_id in ( 1,4 )

V1_ID V1_END      V2_BEGIN         V2_ID
----- ----------- ----------- ----------
    1 02-JAN-2005 01-JAN-2005          1
    1 02-JAN-2005 02-JAN-2005          2
    1 02-JAN-2005 03-JAN-2005          3
    1 02-JAN-2005 04-JAN-2005          4
    1 02-JAN-2005 06-JAN-2005          5
    1 02-JAN-2005 16-JAN-2005          6
    1 02-JAN-2005 17-JAN-2005          7
    1 02-JAN-2005 18-JAN-2005          8
    1 02-JAN-2005 19-JAN-2005          9
    1 02-JAN-2005 21-JAN-2005         10
    1 02-JAN-2005 26-JAN-2005         11
    1 02-JAN-2005 27-JAN-2005         12
    1 02-JAN-2005 28-JAN-2005         13
    1 02-JAN-2005 29-JAN-2005         14
    4 05-JAN-2005 01-JAN-2005          1
    4 05-JAN-2005 02-JAN-2005          2
    4 05-JAN-2005 03-JAN-2005          3
    4 05-JAN-2005 04-JAN-2005          4
    4 05-JAN-2005 06-JAN-2005          5
    4 05-JAN-2005 16-JAN-2005          6
    4 05-JAN-2005 17-JAN-2005          7
    4 05-JAN-2005 18-JAN-2005          8
    4 05-JAN-2005 19-JAN-2005          9
    4 05-JAN-2005 21-JAN-2005         10
    4 05-JAN-2005 26-JAN-2005         11
    4 05-JAN-2005 27-JAN-2005         12
    4 05-JAN-2005 28-JAN-2005         13
    4 05-JAN-2005 29-JAN-2005         14
```

仔细检查以上结果集，我们就会明白为什么 PROJ_ID 1 被包含在最终结果集中，而 PROJ_ID4 却没有。这是因为对于 V1_ID 4 而言，不存在与 V1_END 值相等的 V2_BEGIN。

如果改变一下筛选条件，PROJ_ID 4 也可以被认为是相邻的项目。考虑如下所示的结果集。

```
select *
  from V
 where proj_id <= 5

PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2005 02-JAN-2005
      2 02-JAN-2005 03-JAN-2005
      3 03-JAN-2005 04-JAN-2005
      4 04-JAN-2005 05-JAN-2005
      5 06-JAN-2005 07-JAN-2005
```

如果“相邻的项目”是指其开始日期与另一个项目的结束日期相同，那么 PROJ_ID 4 也应该被包含在结果集里。按照最初的筛选条件，由于前向比较（比较 PROJ_END 和下一行的 PROJ_START）的存在，PROJ_ID 4 被排除在外了，但如果做后向比较（比较 PROJ_START 和前一行的 PROJ_END），那么 PROJ_ID 4 就应该被包含在结果集中。

不妨修改一下上述解决方案，把 PROJ_ID 4 也包含进去：只需增加一个筛选条件，确保 PROJ_START 和 PROJ_END 的相邻关系都被纳入检查范围就行了，而不是仅仅检查 PROJ_END。下面的查询展示了上述改动，该查询产生了一个包含 PROJ_ID 4 的结果集（DISTINCT 是必要的，因为一些行同时满足两个条件）。

```
select distinct
       v1.proj_id,
       v1.proj_start,
       v1.proj_end
  from V v1, V v2
 where v1.proj_end   = v2.proj_start
    or v1.proj_start = v2.proj_end

PROJ_ID PROJ_START  PROJ_END
------ ----------- -----------
     1 01-JAN-2005 02-JAN-2005
     2 02-JAN-2005 03-JAN-2005
     3 03-JAN-2005 04-JAN-2005
     4 04-JAN-2005 05-JAN-2005
```

Oracle

上述自连接方案当然适用于本问题，不过窗口函数 LEAD OVER 用于解决这一类问题更合适。LEAD OVER 函数可以不执行自连接就能查看其他行的数据（尽管该函数要求必须对结果集进行排序）。对于 PROJ_ID1 和 PROJ_ID4，考虑内嵌视图（第 3 ～ 5 行）的结果集。

```
select *
  from (
select proj_id,proj_start,proj_end,
       lead(proj_start)over(order by proj_id) next_proj_start
  from v
       )
 where proj_id in ( 1,4 )

PROJ_ID PROJ_START  PROJ_END    NEXT_PROJ_START
------- ----------- ----------- ---------------
      1 01-JAN-2005 02-JAN-2005 02-JAN-2005
      4 04-JAN-2005 05-JAN-2005 06-JAN-2005
```

仔细检查上面的代码片段及其结果集，不难理解为什么 PROJ_ID 4 会被从最终结果集里排除掉。这是因为 PROJ_END 的日期 2005 年 1 月 5 日与下一个项目的开始日期 2005 年 1 月 6 日不同。

LEAD OVER 函数用于解决这一类问题非常方便，尤其在只遍历部分结果的时候。使用窗口函数时，要记得它们在 FROM 和 WHERE 子句之后才会被评估，因此前面的查询里的 LEAD OVER 函数必须被放入一个内嵌视图才行。否则，LEAD OVER 函数的作用对象就变成了只含有 PROJ_ID 1 和 PROJ_ID 4 的结果集，因为执行过 WHERE 子句之后，其他行都会被过滤掉。

现在，如果改变一下筛选条件，我们也能把 PROJ_ID 4 纳入最终的结果集。考虑视图 V 的前 5 行数据。

```
select *
  from V
 where proj_id <= 5

PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2005 02-JAN-2005
      2 02-JAN-2005 03-JAN-2005
      3 03-JAN-2005 04-JAN-2005
      4 04-JAN-2005 05-JAN-2005
      5 06-JAN-2005 07-JAN-2005
```

如果我们认为 PROJ_ID 4 事实上是相邻的（因为 PROJ_ID 4 的 PROJ_START 等于 PROJ_ID 3 的 PROJ_END），而且只有 PROJ_ID 5 应该被排除在外，那么前述解决方案就不正确了，至少是不全面的。

```
select proj_id,proj_start,proj_end
  from (
select proj_id,proj_start,proj_end,
       lead(proj_start)over(order by proj_id) next_start
  from V
 where proj_id <= 5
       )
 where proj_end = next_start

PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2005 02-JAN-2005
      2 02-JAN-2005 03-JAN-2005
      3 03-JAN-2005 04-JAN-2005
```

如果我们认为 PROJ_ID 4 应该被包含进来，只需额外加上 LAG OVER 函数，并在 WHERE 子句中也增加一个过滤条件即可。

```
select proj_id,proj_start,proj_end
  from (
select proj_id,proj_start,proj_end,
       lead(proj_start)over(order by proj_id) next_start,
       lag(proj_end)over(order by proj_id) last_end
  from V
 where proj_id <= 5
       )
 where proj_end   = next_start
    or proj_start = last_end

PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2005 02-JAN-2005
      2 02-JAN-2005 03-JAN-2005
      3 03-JAN-2005 04-JAN-2005
      4 04-JAN-2005 05-JAN-2005
```

这样一来，PROJ_ID 4 也被纳入最终结果集了，只有 PROJ_ID 5 会被排除在外。当然，如果真的要对本解决方案做出上述代码改动，一定仔细确认需求。

10.2 计算同一组或分区的行之间的差

1. 问题

你想返回每个员工的 DEPTNO、ENAME 和 SAL，以及同一个部门（即 DEPTNO 相同）里不同员工之间的工资差距。工资差距指的是当前员工的 SAL 和入职日期紧随其后的那个员工的 SAL 之间的差值（其实你希望以部门为单位考察资历和工资是否存在相关性）。对于一个部门里入职日期最晚的那个员工，将其工资差距设置为 N/A。最终结果集应该如下所示。

```
DEPTNO ENAME             SAL HIREDATE    DIFF
------ ---------- ---------- ----------- ----------
    10 CLARK            2450 09-JUN-1981      -2550
    10 KING             5000 17-NOV-1981       3700
    10 MILLER           1300 23-JAN-1982        N/A
    20 SMITH             800 17-DEC-1980      -2175
    20 JONES            2975 02-APR-1981        -25
    20 FORD             3000 03-DEC-1981          0
    20 SCOTT            3000 09-DEC-1982       1900
    20 ADAMS            1100 12-JAN-1983        N/A
    30 ALLEN            1600 20-FEB-1981        350
    30 WARD             1250 22-FEB-1981      -1600
    30 BLAKE            2850 01-MAY-1981       1350
    30 TURNER           1500 08-SEP-1981        250
    30 MARTIN           1250 28-SEP-1981        300
    30 JAMES             950 03-DEC-1981        N/A
```

2. 解决方案

这是说明 Oracle 窗口函数 LEAD OVER 和 LAG OVER 的便利性的另一个例子。不需要做额外的连接查询，我们就能方便地查看下一行或者前一行数据。对于其他关系数据库管理系统，可以使用标量子查询，尽管不是那么便利。对于本问题而言，当被迫要用标量子查询或自连接来解决问题时，解决方案就没有那么简单。

DB2、MySQL、PostgreSQL 和 SQL Server

用标量子查询取出紧随当前员工之后入职的员工的 HIREDATE，然后再用另一个标量子查询找出该员工的工资。

```
 1 select deptno,ename,hiredate,sal,
 2        coalesce(cast(sal-next_sal as char(10)),'N/A') as diff
 3   from (
 4 select e.deptno,
 5        e.ename,
 6        e.hiredate,
 7        e.sal,
 8        (select min(sal) from emp d
 9          where d.deptno=e.deptno
10            and d.hiredate =
11                 (select min(hiredate) from emp d
12                   where e.deptno=d.deptno
```

```
13                    and d.hiredate > e.hiredate)) as next_sal
14    from emp e
15         ) x
```

Oracle

使用窗口函数 LEAD OVER 读取与当前行相关的下一个员工的工资。

```
1 select deptno,ename,sal,hiredate,
2        lpad(nvl(to_char(sal-next_sal),'N/A'),10) diff
3   from (
4 select deptno,ename,sal,hiredate,
5        lead(sal)over(partition by deptno
6                          order by hiredate) next_sal
7   from emp
8        )
```

3. 讨论

DB2、MySQL、PostgreSQL 和 SQL Server

首先使用标量子查询找出同一个部门里紧随当前员工之后入职的员工的 HIREDATE，本解决方案在标量子查询中使用了 MIN(HIREDATE) 来确保仅返回一个值，即使同一天入职的员工不止一个人，也只会返回一个值。

```
select e.deptno,
       e.ename,
       e.hiredate,
       e.sal,
       (select min(hiredate) from emp d
          where e.deptno=d.deptno
            and d.hiredate > e.hiredate) as next_hire
  from emp e
 order by 1

DEPTNO ENAME      HIREDATE           SAL NEXT_HIRE
------ ---------- ----------- ---------- -----------
    10 CLARK      09-JUN-1981       2450 17-NOV-1981
    10 KING       17-NOV-1981       5000 23-JAN-1982
    10 MILLER     23-JAN-1982       1300
    20 SMITH      17-DEC-1980        800 02-APR-1981
    20 ADAMS      12-JAN-1983       1100
    20 FORD       03-DEC-1981       3000 09-DEC-1982
    20 SCOTT      09-DEC-1982       3000 12-JAN-1983
    20 JONES      02-APR-1981       2975 03-DEC-1981
    30 ALLEN      20-FEB-1981       1600 22-FEB-1981
    30 BLAKE      01-MAY-1981       2850 08-SEP-1981
    30 MARTIN     28-SEP-1981       1250 03-DEC-1981
    30 JAMES      03-DEC-1981        950
    30 TURNER     08-SEP-1981       1500 28-SEP-1981
    30 WARD       22-FEB-1981       1250 01-MAY-1981
```

然后，使用另一个标量子查询来找出入职日期等于 NEXT_HIRE 的员工的工资。同样，本解决方案使用 MIN 函数来确保只返回一个值。

```
select e.deptno,
       e.ename,
       e.hiredate,
       e.sal,
       (select min(sal) from emp d
         where d.deptno=e.deptno
           and d.hiredate =
                (select min(hiredate) from emp d
                  where e.deptno=d.deptno
                    and d.hiredate > e.hiredate)) as next_sal
  from emp e
 order by 1

DEPTNO ENAME      HIREDATE           SAL   NEXT_SAL
------ ---------- ----------- ---------- ----------
    10 CLARK      09-JUN-1981       2450       5000
    10 KING       17-NOV-1981       5000       1300
    10 MILLER     23-JAN-1982       1300
    20 SMITH      17-DEC-1980        800       2975
    20 ADAMS      12-JAN-1983       1100
    20 FORD       03-DEC-1981       3000       3000
    20 SCOTT      09-DEC-1982       3000       1100
    20 JONES      02-APR-1981       2975       3000
    30 ALLEN      20-FEB-1981       1600       1250
    30 BLAKE      01-MAY-1981       2850       1500
    30 MARTIN     28-SEP-1981       1250        950
    30 JAMES      03-DEC-1981        950
    30 TURNER     08-SEP-1981       1500       1250
    30 WARD       22-FEB-1981       1250       2850
```

最后，计算出 SAL 和 NEXT_SAL 之间的差，并且使用 COALESCE 函数在适当的时候返回 N/A。因为减法运算的结果既有可能是数字，也有可能是 Null，所以必须将其转换为字符串，以便 COALESCE 函数可以正常运行。

```
select deptno,ename,hiredate,sal,
       coalesce(cast(sal-next_sal as char(10)),'N/A') as diff
  from (
select e.deptno,
       e.ename,
       e.hiredate,
       e.sal,
       (select min(sal) from emp d
         where d.deptno=e.deptno
           and d.hiredate =
                (select min(hiredate) from emp d
                  where e.deptno=d.deptno
                    and d.hiredate > e.hiredate)) as next_sal
  from emp e
       ) x
 order by 1

DEPTNO ENAME      HIREDATE            SAL DIFF
------ ---------- ----------- ---------- ---------
    10 CLARK      09-JUN-1981       2450 -2550
```

```
    10 KING       17-NOV-1981       5000 3700
    10 MILLER     23-JAN-1982       1300 N/A
    20 SMITH      17-DEC-1980        800 -2175
    20 ADAMS      12-JAN-1983       1100 N/A
    20 FORD       03-DEC-1981       3000 0
    20 SCOTT      09-DEC-1982       3000 1900
    20 JONES      02-APR-1981       2975 -25
    30 ALLEN      20-FEB-1981       1600 350
    30 BLAKE      01-MAY-1981       2850 1350
    30 MARTIN     28-SEP-1981       1250 300
    30 JAMES      03-DEC-1981        950 N/A
    30 TURNER     08-SEP-1981       1500 250
    30 WARD       22-FEB-1981       1250 -1600
```



本解决方案使用了 MIN(SAL) 函数，这说明我们在某些情况下可能会不知不觉间把一些额外的业务逻辑引入查询，而我们却认为这只是一个纯粹的技术决策。如果一个给定的日期对应多个可能的工资值，我们应该选择最小值、最大值还是平均值呢？本例选择了最小值。在实际工作中，我可能更愿意将选择权交给提出报表请求的客户。

Oracle

首先使用窗口函数 LEAD OVER 为每个员工找出同部门中的“下一个”工资值。对于每个部门里最迟入职的员工，其 NEXT_SAL 列会是 Null。

```
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno order by hiredate) next_sal
  from emp

DEPTNO ENAME             SAL HIREDATE      NEXT_SAL
------ ---------- ---------- ----------- ----------
    10 CLARK            2450 09-JUN-1981       5000
    10 KING             5000 17-NOV-1981       1300
    10 MILLER           1300 23-JAN-1982
    20 SMITH             800 17-DEC-1980       2975
    20 JONES            2975 02-APR-1981       3000
    20 FORD             3000 03-DEC-1981       3000
    20 SCOTT            3000 09-DEC-1982       1100
    20 ADAMS            1100 12-JAN-1983
    30 ALLEN            1600 20-FEB-1981       1250
    30 WARD             1250 22-FEB-1981       2850
    30 BLAKE            2850 01-MAY-1981       1500
    30 TURNER           1500 08-SEP-1981       1250
    30 MARTIN           1250 28-SEP-1981        950
    30 JAMES             950 03-DEC-1981
```

然后，计算出同一个部门里每个员工与紧随其后入职的员工的工资差值。

```
select deptno,ename,sal,hiredate, sal-next_sal diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno order by hiredate) next_sal
  from emp
```

```
       )

DEPTNO ENAME             SAL HIREDATE          DIFF
------ ---------- ---------- ----------- ----------
    10 CLARK            2450 09-JUN-1981      -2550
    10 KING             5000 17-NOV-1981       3700
    10 MILLER           1300 23-JAN-1982
    20 SMITH             800 17-DEC-1980      -2175
    20 JONES            2975 02-APR-1981        -25
    20 FORD             3000 03-DEC-1981          0
    20 SCOTT            3000 09-DEC-1982       1900
    20 ADAMS            1100 12-JAN-1983
    30 ALLEN            1600 20-FEB-1981        350
    30 WARD             1250 22-FEB-1981      -1600
    30 BLAKE            2850 01-MAY-1981       1350
    30 TURNER           1500 08-SEP-1981        250
    30 MARTIN           1250 28-SEP-1981        300
    30 JAMES             950 03-DEC-1981
```

接下来，调用 NVL 函数，在 DIFF 等于 Null 时返回 N/A。为了做到这一点，必须先把 DIFF 值转换为字符串，否则 NVL 函数会执行失败。

```
select deptno,ename,sal,hiredate,
       nvl(to_char(sal-next_sal),'N/A') diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno order by hiredate) next_sal
  from emp
       )

DEPTNO ENAME             SAL HIREDATE    DIFF
------ ---------- ---------- ----------- ---------------
    10 CLARK            2450 09-JUN-1981 -2550
    10 KING             5000 17-NOV-1981 3700
    10 MILLER           1300 23-JAN-1982 N/A
    20 SMITH             800 17-DEC-1980 -2175
    20 JONES            2975 02-APR-1981 -25
    20 FORD             3000 03-DEC-1981 0
    20 SCOTT            3000 09-DEC-1982 1900
    20 ADAMS            1100 12-JAN-1983 N/A
    30 ALLEN            1600 20-FEB-1981 350
    30 WARD             1250 22-FEB-1981 -1600
    30 BLAKE            2850 01-MAY-1981 1350
    30 TURNER           1500 08-SEP-1981 250
    30 MARTIN           1250 28-SEP-1981 300
    30 JAMES             950 03-DEC-1981 N/A
```

最后，调用函数 LPAD 对 DIFF 值进行格式化。这是因为在默认情况下，数字是右对齐的，而字符串则是左对齐的。调用 LPAD 能让所有结果值都变成右对齐。

```
select deptno,ename,sal,hiredate,
       lpad(nvl(to_char(sal-next_sal),'N/A'),10) diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno order by hiredate) next_sal
```

```
  from emp
       )

DEPTNO ENAME             SAL HIREDATE    DIFF
------ ---------- ---------- ----------- ----------
    10 CLARK            2450 09-JUN-1981      -2550
    10 KING             5000 17-NOV-1981       3700
    10 MILLER           1300 23-JAN-1982        N/A
    20 SMITH             800 17-DEC-1980      -2175
    20 JONES            2975 02-APR-1981        -25
    20 FORD             3000 03-DEC-1981          0
    20 SCOTT            3000 09-DEC-1982       1900
    20 ADAMS            1100 12-JAN-1983        N/A
    30 ALLEN            1600 20-FEB-1981        350
    30 WARD             1250 22-FEB-1981      -1600
    30 BLAKE            2850 01-MAY-1981       1350
    30 TURNER           1500 08-SEP-1981        250
    30 MARTIN           1250 28-SEP-1981        300
    30 JAMES             950 03-DEC-1981        N/A
```

本书的大部分实例都没有讨论“特例”（这是考虑到代码的可读性，也有利于我在写作过程中保持思路清晰），但是本例有必要讨论一下在使用 Oracle 的 LEAD OVER 函数时需要注意的重复项问题。对于 EMP 表里那些简单的示例数据而言，并不存在 HIREDATE 相同的行，这也是非常可能发生的状况。因而，正常情况下，我不会讨论如数据重复之类的特例（因为 EMP 表里并没有重复项）。但是，一旦涉及 LEAD 函数，有些读者很可能无法马上想到这一层（对于那些没有 Oracle 经验的读者尤其如此）。考虑如下所示的查询，该查询返回部门编号为 10 的员工之间的工资差距（按照入职先后排序计算出前后两人的工资差值）。

```
select deptno,ename,sal,hiredate,
       lpad(nvl(to_char(sal-next_sal),'N/A'),10) diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno
                         order by hiredate) next_sal
  from emp
 where deptno=10 and empno > 10
       )

DEPTNO ENAME    SAL HIREDATE    DIFF
------ ------ ----- ----------- ----------
    10 CLARK   2450 09-JUN-1981      -2550
    10 KING    5000 17-NOV-1981       3700
    10 MILLER  1300 23-JAN-1982        N/A
```

上述解决方案对于 EMP 表的现有数据而言毫无问题，但如果考虑重复行，就不对了。考虑如下所示的例子，有另外 4 人和员工 KING 同一天入职。

```
insert into emp (empno,ename,deptno,sal,hiredate)
values (1,'ant',10,1000,to_date('17-NOV-1981'))

insert into emp (empno,ename,deptno,sal,hiredate)
values (2,'joe',10,1500,to_date('17-NOV-1981'))
```

```
insert into emp (empno,ename,deptno,sal,hiredate)
values (3,'jim',10,1600,to_date('17-NOV-1981'))

insert into emp (empno,ename,deptno,sal,hiredate)
values (4,'jon',10,1700,to_date('17-NOV-1981'))

select deptno,ename,sal,hiredate,
       lpad(nvl(to_char(sal-next_sal),'N/A'),10) diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno
                         order by hiredate) next_sal
  from emp
 where deptno=10
       )

DEPTNO ENAME    SAL HIREDATE    DIFF
------ ------ ----- ----------- ----------
    10 CLARK   2450 09-JUN-1981       1450
    10 ant     1000 17-NOV-1981       -500
    10 joe     1500 17-NOV-1981      -3500
    10 KING    5000 17-NOV-1981       3400
    10 jim     1600 17-NOV-1981       -100
    10 jon     1700 17-NOV-1981        400
    10 MILLER  1300 23-JAN-1982        N/A
```

我们看到，除了 JON 以外，其他在同一天（11 月 17 日）入职的员工竟然都在和另一个同时入职的人做工资比较！这是不正确的。所有在 11 月 17 日入职的员工都应该和 MILLER 做比较。以员工 ANT 为例，ANT 的 DIFF 值是 −500，这是因为其 SAL 值相较于 JOE 少 500。事实上，ANT 的 DIFF 值应该是 −300 才对，因为其 SAL 值比 MILLER 的少 300，依据 HIREDATE 的顺序，MILLER 才是紧随 ANT 之后入职的员工。上述查询结果的错误应该归因于 LEAD OVER 函数的默认行为方式。在默认情况下，LEAD OVER 函数只往前看一行。因此，对于员工 ANT 而言，基于 HIREDATE 的下一个 SAL 值是 JOE 的 SAL，因为 LEAD OVER 函数只会看下一行，并不会跳过重复项。幸运的是，Oracle 考虑到了这种情况，并允许我们通过传递一个额外的参数告诉 LEAD OVER 函数应该往前看几行。对于本例而言，只需要做一个计数：找出每一个在 11 月 17 日入职的员工和 1 月 23 日（MILLER 的 HIREDATE）之间的距离。下面的解决方案展示了如何实现这一点。

```
select deptno,ename,sal,hiredate,
       lpad(nvl(to_char(sal-next_sal),'N/A'),10) diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal,cnt-rn+1)over(partition by deptno
                         order by hiredate) next_sal
  from (
select deptno,ename,sal,hiredate,
       count(*)over(partition by deptno,hiredate) cnt,
       row_number()over(partition by deptno,hiredate order by sal) rn
  from emp
 where deptno=10
       )
       )
```

```
       )

DEPTNO ENAME    SAL HIREDATE    DIFF
------ ------ ----- ----------- ----------
    10 CLARK   2450 09-JUN-1981       1450
    10 ant     1000 17-NOV-1981       -300
    10 joe     1500 17-NOV-1981        200
    10 jim     1600 17-NOV-1981        300
    10 jon     1700 17-NOV-1981        400
    10 KING    5000 17-NOV-1981       3700
    10 MILLER  1300 23-JAN-1982        N/A
```

现在的解决方案是正确的了。我们看到，所有在 11 月 17 日入职的员工都改为和 MILLER 做比较了。从查询结果里可以看到，现在员工 ANT 的 DIFF 值是 −300，这正是我们希望的结果。你可能不理解传递给 LEAD OVER 的表达式；其实，CNT-RN+1 代表每一个在 11 月 17 日入职的员工到 MILLER 的距离。如下所示的内嵌视图展示了 CNT 和 RN 的值。

```
select deptno,ename,sal,hiredate,
       count(*)over(partition by deptno,hiredate) cnt,
       row_number()over(partition by deptno,hiredate order by sal) rn
  from emp
 where deptno=10

DEPTNO ENAME    SAL HIREDATE           CNT         RN
------ ------ ----- ----------- ---------- ----------
    10 CLARK   2450 09-JUN-1981          1          1
    10 ant     1000 17-NOV-1981          5          1
    10 joe     1500 17-NOV-1981          5          2
    10 jim     1600 17-NOV-1981          5          3
    10 jon     1700 17-NOV-1981          5          4
    10 KING    5000 17-NOV-1981          5          5
    10 MILLER  1300 23-JAN-1982          1          1
```

对于每一个员工而言，CNT 值表示有多少个相同的 HIREDATE，RN 值代表的是部门编号为 10 的员工的排名。排名基于 DEPTNO 和 HIREDATE 分组，因此只有在 HIREDATE 重复的时候，员工的排名才可能大于 1。排名基于 SAL 值排序（并不是必须这么做；只是基于 SAL 值排序比较方便，当然也可以选 EMPNO）。现在我们知道了有多少个重复项，以及每一个重复项对应的排名，它们到 MILLER 的距离就是重复项的数目减去当前排名，然后再加 1（CNT-RN+1）。距离计算的结果以及它对 LEAD OVER 的效果显示如下。

```
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno
                         order by hiredate) incorrect,
       cnt-rn+1 distance,
       lead(sal,cnt-rn+1)over(partition by deptno
                         order by hiredate) correct
  from (
select deptno,ename,sal,hiredate,
       count(*)over(partition by deptno,hiredate) cnt,
       row_number()over(partition by deptno,hiredate
                           order by sal) rn
  from emp
 where deptno=10
```

```
       )

DEPTNO ENAME    SAL HIREDATE     INCORRECT   DISTANCE    CORRECT
------ ------ ----- ----------- ---------- ---------- ----------
    10 CLARK   2450 09-JUN-1981       1000          1       1000
    10 ant     1000 17-NOV-1981       1500          5       1300
    10 joe     1500 17-NOV-1981       1600          4       1300
    10 jim     1600 17-NOV-1981       1700          3       1300
    10 jon     1700 17-NOV-1981       5000          2       1300
    10 KING    5000 17-NOV-1981       1300          1       1300
    10 MILLER  1300 23-JAN-1982                     1
```

现在我们清楚地看到了当传递正确的距离值给 LEAD OVER 函数时会得到什么样的结果。INCORRECT 列代表了传递默认距离 1 给 LEAD OVER 函数时得到的返回值。CORRECT 列代表了为每个重复的 HIREDATE 对应的员工传递到 MILLER 的实际值给 LEAD OVER 函数时得到的返回值。至此，剩下要做的就是为每一行找出 CORRECT 和 SAL 之间的差值，这在前面已经介绍过了。

10.3 定位连续值区间的开始值和结束值

1. 问题

本实例是本章第一个实例的引申，它们都用到了视图 V。在之前的实例中，你已经找到了包含一组连续值的区间，你希望知道它们的开始值和结束值。与本章的第一个实例不同，如果有一行并不属于某个连续值区间，你仍然希望返回它。为什么？这是因为，这样的行自成一个区间，它也有自己的开始值和结束值。视图 V 的数据如下所示。

```
select *
  from V

PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2005 02-JAN-2005
      2 02-JAN-2005 03-JAN-2005
      3 03-JAN-2005 04-JAN-2005
      4 04-JAN-2005 05-JAN-2005
      5 06-JAN-2005 07-JAN-2005
      6 16-JAN-2005 17-JAN-2005
      7 17-JAN-2005 18-JAN-2005
      8 18-JAN-2005 19-JAN-2005
      9 19-JAN-2005 20-JAN-2005
     10 21-JAN-2005 22-JAN-2005
     11 26-JAN-2005 27-JAN-2005
     12 27-JAN-2005 28-JAN-2005
     13 28-JAN-2005 29-JAN-2005
     14 29-JAN-2005 30-JAN-2005
```

你最终希望得到如下所示的结果集。

```
PROJ_GRP PROJ_START  PROJ_END
-------- ----------- -----------
       1 01-JAN-2005 05-JAN-2005
```

```
         2 06-JAN-2005 07-JAN-2005
         3 16-JAN-2005 20-JAN-2005
         4 21-JAN-2005 22-JAN-2005
         5 26-JAN-2005 30-JAN-2005
```

2. 解决方案

相较于本章第一个实例，本问题更为复杂。首先，必须明确什么是区间。PROJ_START 和 PROJ_END 的值决定哪些行属于同一个区间。如果某一行的 PROJ_START 值等于上一行的 PROJ_END 值，那么该行就是“连续”的，或者说它属于某个组。如果某一行的 PROJ_START 值不等于上一行的 PROJ_END 值，并且它的 PROJ_END 值也不等于下一行的 PROJ_START 值，那么该行自身就构成了一个独立的组。识别出区间之后，还要对每个区间相关的行进行分组，并找出每一组的开始值和结束值。

我们来看看最终结果集的第一行数据。PROJ_START 值是视图 V 里 PROJ_ID 1 对应的 PROJ_START，PROJ_END 则是视图 V 里 PROJ_ID 4 对应的 PROJ_END。PROJ_ID 4 后面并没有再出现一个连续值，因此它作为这一组连续值的最后一个被纳入了第一组。

DB2、MySQL、PostgreSQL 和 SQL Server

这些数据库对应的解决方案需要用到视图 V2，以增强代码的可读性。视图 V2 的定义如下。

```
create view v2
as
select a.*,
       case
         when (
            select b.proj_id
              from V b
             where a.proj_start = b.proj_end
              )
              is not null then 0 else 1
       end as flag
  from V a
```

其结果集如下所示。

```
select *
  from V2

PROJ_ID PROJ_START  PROJ_END          FLAG
------- ----------- ----------- ----------
      1 01-JAN-2005 02-JAN-2005          1
      2 02-JAN-2005 03-JAN-2005          0
      3 03-JAN-2005 04-JAN-2005          0
      4 04-JAN-2005 05-JAN-2005          0
      5 06-JAN-2005 07-JAN-2005          1
      6 16-JAN-2005 17-JAN-2005          1
      7 17-JAN-2005 18-JAN-2005          0
      8 18-JAN-2005 19-JAN-2005          0
      9 19-JAN-2005 20-JAN-2005          0
     10 21-JAN-2005 22-JAN-2005          1
     11 26-JAN-2005 27-JAN-2005          1
     12 27-JAN-2005 28-JAN-2005          0
```

```
     13 28-JAN-2005 29-JAN-2005          0
     14 29-JAN-2005 30-JAN-2005          0
```

在视图 V2 的基础上得到的解决方案如下所示。首先，找出那些属于某个连续值区间的行，并为它们分组。然后，调用 MIN 函数和 MAX 函数找出每一组的开始值和结束值。

```
 1 select proj_grp,
 2        min(proj_start) as proj_start,
 3        max(proj_end) as proj_end
 4   from (
 5 select a.proj_id,a.proj_start,a.proj_end,
 6        (select sum(b.flag)
 7           from V2 b
 8          where b.proj_id <= a.proj_id) as proj_grp
 9   from V2 a
10        ) x
11  group by proj_grp
```

Oracle

上述解决方案当然也适用于 Oracle。不过，借助 Oracle 的窗口函数 LAG OVER，无须额外的视图也能解决本问题。我们可以利用 LAG OVER 函数判定前一行的 PROJ_END 是否等于当前行的 PROJ_START，并以此为标准对当前行进行分组。分组完成之后，接着调用聚合函数 MIN 和 MAX 分别找出每组的开始值和结束值。

```
 1 select proj_grp, min(proj_start), max(proj_end)
 2   from (
 3 select proj_id,proj_start,proj_end,
 4        sum(flag)over(order by proj_id) proj_grp
 5   from (
 6 select proj_id,proj_start,proj_end,
 7        case when
 8             lag(proj_end)over(order by proj_id) = proj_start
 9             then 0 else 1
10        end flag
11   from V
12        )
13        )
14  group by proj_grp
```

3. 讨论

DB2、MySQL、PostgreSQL 和 SQL Server

有了视图 V2，本问题就相对容易一些了。视图 V2 在 CASE 表达式里用了一个标量子查询来判断当前行是否属于某个连续值区间。如果当前行属于某个连续值区间，那么别名为 FLAG 的 CASE 表达式将返回 0；反之，则返回 1（判定当前行是否属于一组连续值区间的方法是：是否有一条记录的 PROJ_END 值等于当前行的 PROJ_START 值）。下一步是逐一查看内嵌视图 X（第 5 ～ 9 行）的查询结果。内嵌视图 X 返回视图 V2 的全部行以及针对 FLAG 的累计值，该累计值就是我们分组的依据，如下所示。

```
select a.proj_id,a.proj_start,a.proj_end,
       (select sum(b.flag)
          from v2 b
```

```
        where b.proj_id <= a.proj_id) as proj_grp
  from v2 a

PROJ_ID PROJ_START  PROJ_END      PROJ_GRP
------- ----------- ----------- ----------
      1 01-JAN-2005 02-JAN-2005          1
      2 02-JAN-2005 03-JAN-2005          1
      3 03-JAN-2005 04-JAN-2005          1
      4 04-JAN-2005 05-JAN-2005          1
      5 06-JAN-2005 07-JAN-2005          2
      6 16-JAN-2005 17-JAN-2005          3
      7 17-JAN-2005 18-JAN-2005          3
      8 18-JAN-2005 19-JAN-2005          3
      9 19-JAN-2005 20-JAN-2005          3
     10 21-JAN-2005 22-JAN-2005          4
     11 26-JAN-2005 27-JAN-2005          5
     12 27-JAN-2005 28-JAN-2005          5
     13 28-JAN-2005 29-JAN-2005          5
     14 29-JAN-2005 30-JAN-2005          5
```

现在已经确定好各个区间了，接下来要针对 PROJ_START 和 PROJ_END 分别调用聚合函数 MIN 和 MAX 找出每个区间的开始值和结束值，然后根据上述累计值分组。

Oracle

对于本实例而言，窗口函数 LAG OVER 非常有用。我们无须使用自连接、标量子查询或额外的视图就能访问前一行的 PROJ_END 值。去掉了 CASE 表达式的 LAG OVER 函数的执行结果显示如下。

```
select proj_id,proj_start,proj_end,
       lag(proj_end)over(order by proj_id) prior_proj_end
  from V

PROJ_ID PROJ_START  PROJ_END    PRIOR_PROJ_END
------- ----------- ----------- --------------
      1 01-JAN-2005 02-JAN-2005
      2 02-JAN-2005 03-JAN-2005 02-JAN-2005
      3 03-JAN-2005 04-JAN-2005 03-JAN-2005
      4 04-JAN-2005 05-JAN-2005 04-JAN-2005
      5 06-JAN-2005 07-JAN-2005 05-JAN-2005
      6 16-JAN-2005 17-JAN-2005 07-JAN-2005
      7 17-JAN-2005 18-JAN-2005 17-JAN-2005
      8 18-JAN-2005 19-JAN-2005 18-JAN-2005
      9 19-JAN-2005 20-JAN-2005 19-JAN-2005
     10 21-JAN-2005 22-JAN-2005 20-JAN-2005
     11 26-JAN-2005 27-JAN-2005 22-JAN-2005
     12 27-JAN-2005 28-JAN-2005 27-JAN-2005
     13 28-JAN-2005 29-JAN-2005 28-JAN-2005
     14 29-JAN-2005 30-JAN-2005 29-JAN-2005
```

从完整的代码中可以看到，CASE 表达式只是比较了 LAG OVER 函数返回的结果和当前行的 PROJ_START 值；如果两个值相等，则返回 0，否则返回 1。下一步就是针对 CASE 表达式返回的 0 和 1 产生一个累计值，从而把每一行都编入某个组。上述累计值的计算结果显示如下。

```
select proj_id,proj_start,proj_end,
       sum(flag)over(order by proj_id) proj_grp
  from (
select proj_id,proj_start,proj_end,
       case when
            lag(proj_end)over(order by proj_id) = proj_start
            then 0 else 1
       end flag
  from V
       )

PROJ_ID PROJ_START  PROJ_END      PROJ_GRP
------- ----------- ----------- ----------
      1 01-JAN-2005 02-JAN-2005          1
      2 02-JAN-2005 03-JAN-2005          1
      3 03-JAN-2005 04-JAN-2005          1
      4 04-JAN-2005 05-JAN-2005          1
      5 06-JAN-2005 07-JAN-2005          2
      6 16-JAN-2005 17-JAN-2005          3
      7 17-JAN-2005 18-JAN-2005          3
      8 18-JAN-2005 19-JAN-2005          3
      9 19-JAN-2005 20-JAN-2005          3
     10 21-JAN-2005 22-JAN-2005          4
     11 26-JAN-2005 27-JAN-2005          5
     12 27-JAN-2005 28-JAN-2005          5
     13 28-JAN-2005 29-JAN-2005          5
     14 29-JAN-2005 30-JAN-2005          5
```

现在每一行都被放入了各自的组，只要针对 PROJ_START 值和 PROJ_END 值分别调用聚合函数 MIN 和 MAX，然后基于 PROJ_GRP 列的累计值分组即可。

10.4 为值区间填充缺失值

1. 问题

你想列出整个 20 世纪 80 年代里每年新入职的员工人数，但有一些年份并没有新增员工。你希望返回如下所示的结果集。

```
YR          CNT
---- ----------
1980          1
1981         10
1982          2
1983          1
1984          0
1985          0
1986          0
1987          0
1988          0
1989          0
```

2. 解决方案

本解决方案的关键之处在于如何为那些没有新增员工的年份返回 0。如果在一个给定的年

份里没有新入职的员工，那么 EMP 表里就不存在对应的行。既然表里不包含这一年，我们该如何为这一年返回计数值 0 呢？本解决方案需要用到外连接操作。我们要拼凑一个包含了所有目标年份的结果集，然后针对 EMP 表执行 COUNT 查询，以判断每一年里是否新增了员工。

DB2

把 EMP 表作为数据透视表（因为它有 14 行数据），并调用内置函数 YEAR，为 20 世纪 80 年代的每一个年份生成一行数据。然后，外连接 EMP 表，并计算每年新增了多少名员工。

```
 1 select x.yr, coalesce(y.cnt,0) cnt
 2   from (
 3 select year(min(hiredate)over()) -
 4        mod(year(min(hiredate)over()),10) +
 5        row_number()over()-1 yr
 6   from emp fetch first 10 rows only
 7        )x
 8   left join
 9        (
10 select year(hiredate) yr1, count(*) cnt
11   from emp
12  group by year(hiredate)
13        ) y
14     on ( x.yr = y.yr1 )
```

Oracle

把 EMP 表作为数据透视表（因为它有 14 行数据），并调用内置函数 TO_NUMBER 和 TO_CHAR，为 20 世纪 80 年代的每一个年份生成一行数据。然后，外连接 EMP 表，并计算每年新增了多少名员工。

```
 1  select x.yr, coalesce(cnt,0) cnt
 2    from (
 3  select extract(year from min(hiredate)over()) -
 4         mod(extract(year from min(hiredate)over()),10) +
 5         rownum-1 yr
 6    from emp
 7   where rownum <= 10
 8         ) x,
 9         (
10  select to_number(to_char(hiredate,'YYYY')) yr, count(*) cnt
11    from emp
12   group by to_number(to_char(hiredate,'YYYY'))
13         ) y
14   where x.yr = y.yr(+)
```

如果使用的是 Oracle 9*i* 及后续版本，则不妨使用新提供的 JOIN 子句。

```
 1  select x.yr, coalesce(cnt,0) cnt
 2    from (
 3  select extract(year from min(hiredate)over()) -
 4         mod(extract(year from min(hiredate)over()),10) +
 5         rownum-1 yr
 6    from emp
 7   where rownum <= 10
```

```
 8         ) x
 9   left join
10         (
11 select to_number(to_char(hiredate,'YYYY')) yr, count(*) cnt
12   from emp
13  group by to_number(to_char(hiredate,'YYYY'))
14         ) y
15     on ( x.yr = y.yr )
```

PostgreSQL 和 MySQL

把 T10 表作为数据透视表（因为它有 10 行数据），并调用内置函数 EXTRACT，为 20 世纪 80 年代的每一个年份生成一行数据。然后，外连接 EMP 表，并计算每年新增了多少名员工。

```
 1 select y.yr, coalesce(x.cnt,0) as cn
 2   from (
 3 select min_year-mod(cast(min_year as int),10)+rn as yr
 4   from (
 5 select (select min(extract(year from hiredate))
 6           from emp) as min_year,
 7         id-1 as rn
 8   from t10
 9         ) a
10         ) y
11   left join
12         (
13 select extract(year from hiredate) as yr, count(*) as cnt
14   from emp
15  group by extract(year from hiredate)
16         ) x
17     on ( y.yr = x.yr )
```

SQL Server

把 EMP 表作为数据透视表（因为它有 14 行数据），并调用内置函数 YEAR，为 20 世纪 80 年代的每一个年份生成一行数据。然后，外连接 EMP 表，并计算每年新增了多少名员工。

```
 1 select x.yr, coalesce(y.cnt,0) cnt
 2   from (
 3 select top (10)
 4         (year(min(hiredate)over()) -
 5          year(min(hiredate)over( ))%10)+
 6          row_number()over(order by hiredate)-1 yr
 7   from emp
 8         ) x
 9   left join
10         (
11 select year(hiredate) yr, count(*) cnt
12   from emp
13  group by year(hiredate)
14         ) y
15     on ( x.yr = y.yr )
```

3. 讨论

尽管各个数据库的解决方案在语法上有所不同，做法却并无二致。内嵌视图 X 先找出最早

的 HIREDATE 值对应的年份，进而返回 20 世纪 80 年代的每一年。下一步是用最早的年份减去该年份模 10 计算的结果，然后再加上 RN-1。为了更清楚地了解其工作原理，我们不妨实际运行一下内嵌视图 X，并分别返回其中涉及的每一个值。下面列出了两个版本的内嵌视图 X，它们分别使用窗口函数 MIN OVER（DB2、Oracle 和 SQL Server）和标量子查询（MySQL 和 PostgreSQL）来得到对应的结果集。

```
select year(min(hiredate)over()) -
       mod(year(min(hiredate)over()),10) +
       row_number()over()-1 yr,
       year(min(hiredate)over()) min_year,
       mod(year(min(hiredate)over()),10) mod_yr,
       row_number()over()-1 rn
  from emp fetch first 10 rows only

  YR   MIN_YEAR     MOD_YR         RN
---- ---------- ---------- ----------
1980       1980          0          0
1981       1980          0          1
1982       1980          0          2
1983       1980          0          3
1984       1980          0          4
1985       1980          0          5
1986       1980          0          6
1987       1980          0          7
1988       1980          0          8
1989       1980          0          9

select min_year-mod(min_year,10)+rn as yr,
       min_year,
       mod(min_year,10) as mod_yr
       rn
  from (
select (select min(extract(year from hiredate))
          from emp) as min_year,
       id-1 as rn
  from t10
       ) x

  YR   MIN_YEAR     MOD_YR         RN
---- ---------- ---------- ----------
1980       1980          0          0
1981       1980          0          1
1982       1980          0          2
1983       1980          0          3
1984       1980          0          4
1985       1980          0          5
1986       1980          0          6
1987       1980          0          7
1988       1980          0          8
1989       1980          0          9
```

内嵌视图 Y 返回每一个 HIREDATE 对应的年份，以及那一年新增的员工人数。

```
select year(hiredate) yr, count(*) cnt
  from emp
 group by year(hiredate)

   YR        CNT
----- ----------
 1980          1
 1981         10
 1982          2
 1983          1
```

最后，外连接内嵌视图 Y 到内嵌视图 X，这样即使某一年没有新增员工，也会返回这一年的计数结果。

10.5 生成连续的数值

1. 问题

你希望有一个“行数据生成器”。如果你的查询里需要数据透视表，就用得着这个行数据生成器。例如，你想返回如下所示的结果集，并且可以为它指定任意数目的行数据。

```
ID
---
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
...
```

如果数据库提供了可以动态地生成行数据的内置函数，那么就不需要预先创建一个固定行数的数据透视表。这就是动态的行数据生成器如此有用的原因。否则，我们必须借助一个传统的、行数固定（也许并不够用）的数据透视表来生成所需的行数据。

2. 解决方案

本解决方案展示如何返回从 1 开始递增至 10 的 10 行数据。我们也可以简单地改动一下代码，以返回任意数目的行。

可以返回从 1 开始递增的值，这种能力为其他许多问题的解决方案打开了方便之门。例如，我们可以生成数字，并加上日期值，这样就能生成连续的日期了。也可以借助这些数字来解析字符串。

DB2 和 SQL Server

使用 WITH 递归查询生成一系列含有递增值的行。先借助一个像 T1 这样的只有 1 行数据的表来启动行数据生成操作，其余的交给 WITH 子句即可。

```
 1  with x (id)
 2  as (
 3  select 1
 4    from t1
 5   union all
 6  select id+1
 7    from x
 8   where id+1 <= 10
 9  )
10  select * from x
```

下面是另一个替代方案，只适用于 DB2。该方案的优点是不需要 T1 表。

```
1  with x (id)
2  as (
3  values (1)
4   union all
5  select id+1
6    from x
7  where id+1 <= 10
8 )
9 select * from x
```

Oracle

使用 CONNECT BY 递归查询（适用于 Oracle 9*i* 及后续版本）。如果使用的是 Oracle 9*i*，我们要么把 CONNECT BY 放入一个内嵌视图，要么把它放进 WITH 子句。

```
1  with x
2  as (
3  select level id
4    from dual
5    connect by level <= 10
6  )
7  select * from x
```

对于 Oracle Database 10*g* 及后续版本，则可以用 MODEL 子句生成行数据。

```
1 select array id
2   from dual
3  model
4    dimension by (0 idx)
5    measures(1 array)
6    rules iterate (10) (
7      array[iteration_number] = iteration_number+1
8    )
```

PostgreSQL

使用 GENERATE_SERIES 函数，该函数就是为快速生成行数据而设计的。

```
1 select id
2   from generate_series (1,10) x(id)
```

3. 讨论

DB2 和 SQL Server

WITH 递归查询逐步递增 ID（初始值为 1），直到背离了 WHERE 子句的条件为止。为了开启递归操作，需要先生成第一行数据，这一行里包含的值应该是 1。我们可以通过 SELECT 1 FROM T1 实现这一点，T1 表只包含一行数据；对于 DB2 而言，还可以使用 VALUES 子句生成只含有一行数据的结果集。

Oracle

本解决方案把 CONNECT BY 子查询放进了 WITH 子句。在 WHERE 子句中断之前，行数据会被连续生成出来。Oracle 会自动递增伪列 LEVEL 的值，我们不必再做什么。

在 MODEL 子句解决方案里，有一个显式的 ITERATE 命令，该命令帮助生成多行数据。如果没有 ITERATE 子句，则只返回一行数据，因为 DUAL 表仅包含一行数据。

```
select array id
  from dual
 model
   dimension by (0 idx)
   measures(1 array)
   rules ()

ID
--
 1
```

MODEL 子句不仅能让我们像访问数组一样访问行数据，还允许我们方便地创建新的行或返回表里不存在的行。在本解决方案中，IDX 是数组下标（数组里某个特定值的位置），ARRAY（别名 ID）是行数据构成的“数组”。第一行的默认值是 1，可以通过 ARRAY[0] 来访问。Oracle 提供了 ITERATION_NUMBER 函数，以便我们知道迭代次数。本解决方案迭代了 10 次，因而 ITERATION_NUMBER 从 0 增加到了 9。为每个 ITERATION_NUMBER 值加上 1，结果就是从 1 到 10。

执行以下查询，会更便于我们理解 MODEL 子句的作用。

```
select 'array['||idx||'] = '||array as output
  from dual
 model
   dimension by (0 idx)
   measures(1 array)
   rules iterate (10) (
     array[iteration_number] = iteration_number+1
   )

OUTPUT
------------------
array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5
array[5] = 6
```

```
array[6] = 7
array[7] = 8
array[8] = 9
array[9] = 10
```

PostgreSQL

全部工作都交给 GENERATE_SERIES 函数来完成。该函数有 3 个参数，它们都是数值类型。第一个参数是初始值，第二个参数是结束值，第三个参数是可选项，代表“步长”（每次增加的值）。如果没有指定第 3 个参数，则默认每次增加 1。

GENERATE_SERIES 函数功能强大，我们传递给它的参数甚至可以不是常量。例如，我们希望返回 5 行数据，初始值为 10，结束值为 30，步长为 5，那么就应该会得到如下所示的结果集。

```
 ID
---
 10
 15
 20
 25
 30
```

为了达到上述目的，我们甚至可以写出类似这样的代码。

```
select id
  from generate_series(
         (select min(deptno) from emp),
         (select max(deptno) from emp),
         5
        ) x(id)
```

需要注意的是，我们在编写以上查询代码的时候并不知道要传递给 GENERATE_SERIES 函数的参数值是什么。只有当实际执行主查询的时候，那两个子查询才会把实际的参数值计算出来。

第 11 章

高级查询

毫无疑问，到目前为止本书都在讲查询。我们已经讨论了各种各样的查询，它们使用连接、WHERE 子句和分组等技巧筛选出我们所需要的结果。不过，有一些类型的查询操作不同于其他的查询。有时候我们想一次显示一页结果集。关于这个问题，一方面要查找出我们希望显示的全部结果。另一方面则是在用户浏览查询结果的过程中，我们需要不断地查找下一页要显示的内容。有些人可能认为分页不应该属于查询问题，但它可以被认为是查询问题，并且我们也可以用 SQL 查询的方式来解决这类问题。本章将主要介绍这一类查询的解决方案。

11.1　结果集分页

1. 问题

你想对结果集进行分页，或者“滚动浏览”一组结果集。例如，你希望从 EMP 表返回最前面的 5 条工资记录，然后返回接下来的 5 条，等等。你的目的是让用户一次看到 5 条记录，并在每次点击“下一页”按钮之后变换显示内容。

2. 解决方案

SQL 里并没有“最先”“最后”或“下一个”这样的概念，我们必须对行记录做出明确的排序。只有做过了排序，才有可能准确地从结果集中返回指定区间的记录。

DB2、Oracle 和 SQL Server

使用窗口函数 ROW_NUMBER OVER 实现排序，并且在 WHERE 子句中指定我们希望返回的行。例如，返回第 1 到第 5 行。

```
select sal
  from (
select row_number() over (order by sal) as rn,
```

```
       sal
  from emp
       ) x
 where rn between 1 and 5

 SAL
----
 800
 950
1100
1250
1250
```

然后，返回第 6 行到第 10 行。

```
select sal
  from (
select row_number() over (order by sal) as rn,
       sal
  from emp
       ) x
 where rn between 6 and 10

  SAL
-----
 1300
 1500
 1600
 2450
 2850
```

通过改变 WHERE 子句，我们能返回任意区间内的行。

MySQL 和 PostgreSQL

对于这两种数据库而言，滚动结果集非常容易，因为它们支持 LIMIT 和 OFFSET 子句。使用 LIMIT 子句指定要返回的行数，使用 OFFSET 子句指定要跳过的行数。例如，按照工资排序返回最前面的 5 行。

```
select sal
  from emp
 order by sal limit 5 offset 0

  SAL
------
   800
   950
  1100
  1250
  1250
```

然后，返回接下来的 5 行。

```
select sal
  from emp
 order by sal limit 5 offset 5
```

```
  SAL
-----
 1300
 1500
 1600
 2450
 2850
```

LIMIT 和 OFFSET 子句使得 MySQL 和 PostgreSQL 解决方案的代码变得更简单，而且更具可读性。

3. 讨论

DB2、Oracle 和 SQL Server

内嵌视图 X 里的窗口函数 ROW_NUMBER OVER 将会为每一行工资记录分配一个唯一的数字编号（从 1 开始递增）。下面是内嵌视图 X 的查询结果集。

```
select row_number() over (order by sal) as rn,
       sal
  from emp

RN        SAL
-- ----------
 1        800
 2        950
 3       1100
 4       1250
 5       1250
 6       1300
 7       1500
 8       1600
 9       2450
10       2850
11       2975
12       3000
13       3000
14       5000
```

一旦每一行工资记录都被指定了数字编号，通过指定 RN 的值就可以筛选出我们想要返回的区间。

对于 Oracle 用户来说，有一个替代方案：可以用 ROWNUM 函数来代替 ROW_NUMBER OVER 函数，同样能为每一行记录生成一个序号。

```
select sal
  from (
select sal, rownum rn
  from (
select sal
  from emp
 order by sal
       )
       )
 where rn between 6 and 10
```

```
  SAL
-----
 1300
 1500
 1600
 2450
 2850
```

使用 ROWNUM 的话，就需要多写一层子查询。最内层的子查询对工资进行排序。接下来的外层子查询为每一行分配序号。最后，最外层的 SELECT 返回我们希望显示的数据。

MySQL 和 PostgreSQL

SELECT 里的 OFFSET 子句使得整个查询语句看起来更加直观，更具可读性。OFFSET 等于 0 表示将从第 1 行开始读取，OFFSET 等于 5 表示从第 6 行开始，OFFSET 等于 10 表示从第 11 行开始。LIMIT 子句则限定了返回的记录行数。通过结合这两种子句，我们就能很容易地在结果集中指定从哪一行开始，并同时指定返回多少行。

11.2 跳过*n*行记录

1. 问题

你想用一个查询来隔行返回 EMP 表中的记录；你希望获得第一个员工、第三个员工，等等。例如，从下面的结果集：

```
ENAME
--------
ADAMS
ALLEN
BLAKE
CLARK
FORD
JAMES
JONES
KING
MARTIN
MILLER
SCOTT
SMITH
TURNER
WARD
```

你希望返回的是：

```
ENAME
----------
ADAMS
BLAKE
FORD
JONES
MARTIN
SCOTT
TURNER
```

2. 解决方案

为了从一个结果集中跳过第 2 行、第 4 行或第 *n* 行，我们必须对结果集先排序，否则就没有所谓的“第一个”“下一个”“第二个”或者“第四个”等概念。

DB2、Oracle 和 SQL Server

使用窗口函数 `ROW_NUMBER OVER` 为每一行分配一个序号，这样就可以借助模函数跳过我们不想要的行了。DB2 和 Oracle 的模函数是 `MOD`。SQL Server 则使用 `%` 操作符。下面的例子使用 `MOD` 跳过编号为偶数的行。

```
1  select ename
2    from (
3  select row_number() over (order by ename) rn,
4         ename
5    from emp
6         ) x
7   where mod(rn,2) = 1
```

MySQL 和 PostgreSQL

这两种数据库不提供支持排序或为每一行数据编排序号的内置函数，因而需要使用标量子查询来模拟实现类似功能（本例中根据员工名字排序），然后使用模函数跳过不需要的行。

```
1  select x.ename
2    from (
3  select a.ename,
4         (select count(*)
5            from emp b
6           where b.ename <= a.ename) as rn
7    from emp a
8         )x
9   where mod(x.rn,2) = 1
```

3. 讨论

DB2、Oracle 和 SQL Server

在内嵌视图 `X` 里调用窗口函数 `ROW_NUMBER OVER` 将会为每一行分配一个序号（没有附加任何条件，也不去除重复的姓名），结果显示如下。

```
select row_number() over (order by ename) rn, ename
  from emp

RN ENAME
-- --------
 1 ADAMS
 2 ALLEN
 3 BLAKE
 4 CLARK
 5 FORD
 6 JAMES
 7 JONES
 8 KING
 9 MARTIN
10 MILLER
11 SCOTT
```

```
12 SMITH
13 TURNER
14 WARD
```

最后，只需调用模函数跳过不需要的行即可。

MySQL 和 PostgreSQL

因为没有内置函数帮助我们为每一行编号，我们改用标量子查询为员工的名字编号。内嵌视图 X 实现了为每个名字编号，结果如下所示。

```
select a.ename,
       (select count(*)
          from emp b
         where b.ename <= a.ename) as rn
  from emp a

ENAME              RN
---------- ----------
ADAMS               1
ALLEN               2
BLAKE               3
CLARK               4
FORD                5
JAMES               6
JONES               7
KING                8
MARTIN              9
MILLER             10
SCOTT              11
SMITH              12
TURNER             13
WARD               14
```

最后，针对产生的行编号调用模函数来跳过不需要的行。

11.3 在外连接查询里使用OR逻辑

1. 问题

你想返回部门编号为 10 和 20 的全体员工的名字和部门信息，以及部门编号为 30 和 40 的部门信息（但不包含员工信息）。你最初试图这样做。

```
select e.ename, d.deptno, d.dname, d.loc
  from dept d, emp e
 where d.deptno = e.deptno
   and (e.deptno = 10 or e.deptno = 20)
 order by 2

ENAME       DEPTNO DNAME          LOC
------- ---------- -------------- -------------
CLARK           10 ACCOUNTING     NEW YORK
KING            10 ACCOUNTING     NEW YORK
MILLER          10 ACCOUNTING     NEW YORK
SMITH           20 RESEARCH       DALLAS
```

```
ADAMS            20 RESEARCH       DALLAS
FORD             20 RESEARCH       DALLAS
SCOTT            20 RESEARCH       DALLAS
JONES            20 RESEARCH       DALLAS
```

以上查询里的连接操作是内连接，因此返回的结果集里不包含 DEPTNO 是 30 和 40 的部门信息。

在下面的查询里你又试图将 EMP 表外连接到 DEPT 表，但仍然没有得到正确的结果集。

```
select e.ename, d.deptno, d.dname, d.loc
  from dept d left join emp e
    on (d.deptno = e.deptno)
 where e.deptno = 10
    or e.deptno = 20
 order by 2

ENAME       DEPTNO DNAME          LOC
------- ---------- -------------- -----------
CLARK           10 ACCOUNTING     NEW YORK
KING            10 ACCOUNTING     NEW YORK
MILLER          10 ACCOUNTING     NEW YORK
SMITH           20 RESEARCH       DALLAS
ADAMS           20 RESEARCH       DALLAS
FORD            20 RESEARCH       DALLAS
SCOTT           20 RESEARCH       DALLAS
JONES           20 RESEARCH       DALLAS
```

其实，你只是希望得到如下所示的结果集。

```
ENAME       DEPTNO DNAME          LOC
------- ---------- -------------- ---------
CLARK           10 ACCOUNTING     NEW YORK
KING            10 ACCOUNTING     NEW YORK
MILLER          10 ACCOUNTING     NEW YORK
SMITH           20 RESEARCH       DALLAS
JONES           20 RESEARCH       DALLAS
SCOTT           20 RESEARCH       DALLAS
ADAMS           20 RESEARCH       DALLAS
FORD            20 RESEARCH       DALLAS
                30 SALES          CHICAGO
                40 OPERATIONS     BOSTON
```

2. 解决方案

DB2、MySQL、PostgreSQL 和 SQL Server

把 OR 条件移到 JOIN 子句里。

```
1  select e.ename, d.deptno, d.dname, d.loc
2    from dept d left join emp e
3      on (d.deptno = e.deptno
4          and (e.deptno=10 or e.deptno=20))
5   order by 2
```

或者，我们也可以先利用内嵌视图过滤 EMP.DEPTNO，然后再执行外连接。

```
1  select e.ename, d.deptno, d.dname, d.loc
2    from dept d
3    left join
4         (select ename, deptno
5            from emp
6           where deptno in ( 10, 20 )
7         ) e on ( e.deptno = d.deptno )
8   order by 2
```

Oracle

对于 Oracle 9*i* 及后续版本，上述针对其他数据库的解决方案也适用。除此之外，我们也可以使用 CASE 或 DECODE 的变通方案。下面是使用 CASE 的解决方案。

```
select e.ename, d.deptno, d.dname, d.loc
  from dept d, emp e
 where d.deptno = e.deptno (+)
   and d.deptno = case when e.deptno(+) = 10 then e.deptno(+)
                       when e.deptno(+) = 20 then e.deptno(+)
                  end
 order by 2
```

接下来的解决方案做法也是一样的，但用到了 DECODE 函数。

```
select e.ename, d.deptno, d.dname, d.loc
  from dept d, emp e
 where d.deptno = e.deptno (+)
   and d.deptno = decode(e.deptno(+),10,e.deptno(+),
                                     20,e.deptno(+))
 order by 2
```

如果用了 Oracle 专有的外连接语法“(+)”，并在外连接列上用到了 IN 或 OR 条件，查询操作就会返回错误。解决办法是把 IN 或 OR 条件移到一个内嵌视图中。

```
select e.ename, d.deptno, d.dname, d.loc
  from dept d,
      ( select ename, deptno
          from emp
         where deptno in ( 10, 20 )
       ) e
 where d.deptno = e.deptno (+)
 order by 2
```

3. 讨论

DB2、MySQL、PostgreSQL 和 SQL Server

针对这些数据库，我们提供了两种解决方案。第一种把 OR 条件移到了 JOIN 子句里，使它成为连接条件的一部分。这样一来，我们既能筛选出 EMP 表的数据，又不会丢掉 DEPT 表里 DEPTNO 等于 30 和 40 的数据。

第二种解决方案把过滤条件移到了内嵌视图里。内嵌视图 E 基于 EMP.DEPTNO 过滤数据，从 EMP 表里提取出我们感兴趣的行。然后，这些行被外连接到 DEPT 表。DEPT 表是外连接的基础表，因此包括部门编号为 30 和 40 在内的所有部门都会被返回。

Oracle

旧式的外连接语法似乎存在缺陷，我们使用了 CASE 和 DECODE 函数以避免该问题。除此之外，使用了内嵌视图 E 的那个解决方案，首先从 EMP 表里筛选出我们感兴趣的行，然后再外连接到 DEPT。

11.4　识别互逆的记录

1. 问题

你有一张表，其中包括了两次考试的成绩，你想找出哪些分数互逆的（reciprocal）。我们先看一下下面视图 V 的结果集。

```
select *
  from V

TEST1      TEST2
----- ----------
   20         20
   50         25
   20         20
   60         30
   70         90
   80        130
   90         70
  100         50
  110         55
  120         60
  130         80
  140         70
```

仔细看上述结果，可以发现 TEST1 等于 70 的考试分数和 TEST2 等于 90 的考试分数是互逆的（TEST1 存在一个 90 的分数，并且 TEST2 也存在一个 70 的分数）。同样地，TEST1 等于 80 的分数和 TEST2 等于 130 的分数也是互逆的，因为 TEST1 里有 130，TEST2 里也有 80。并且，TEST1 等于 20 的分数和 TEST2 等于 20 的分数也是互逆的，因为同样存在 TEST2 等于 20、并且 TEST1 等于 20 的记录。你只希望识别出互逆的记录的集合。因此，你希望得到的结果集如下所示。

```
TEST1      TEST2
----- ----------
   20         20
   70         90
   80        130
```

而不是下面的这个。

```
TEST1      TEST2
----- ----------
   20         20
   20         20
   70         90
   80        130
   90         70
  130         80
```

2. 解决方案

使用自连接识别出 TEST1 等于 TEST2，并且 TEST2 等于 TEST1 的那些行。

```
select distinct v1.*
  from V v1, V v2
 where v1.test1  = v2.test2
   and v1.test2  = v2.test1
   and v1.test1 <= v1.test2
```

3. 讨论

自连接产生了一组笛卡儿积，这样一个 TEST1 分数可以和每一个 TEST2 分数进行比较；反之，一个 TEST2 分数也可以和每一个 TEST1 分数进行比较。下面的查询将会识别出互逆的分数。

```
select v1.*
  from V v1, V v2
 where v1.test1 = v2.test2
   and v1.test2 = v2.test1

TEST1      TEST2
----- ----------
   20         20
   20         20
   20         20
   20         20
   90         70
  130         80
   70         90
   80        130
```

使用 DISTINCT 能确保从最后的结果集里删除掉重复项。WHERE 子句的最后一个过滤条件（and V1.TEST1 <= V1.TEST2）将确保只返回 TEST1 小于或等于 TEST2 的那一对互逆的分数。

11.5　提取最靠前的*n*行记录

1. 问题

你想基于某种排序方式从结果集中提取出限定数目的记录。例如，你希望返回 5 个工资最高的员工的姓名和工资。

2. 解决方案

本解决方案的关键之处有两点：首先要基于我们感兴趣的列对数据集进行排序，然后从结果集里提取出所需数目的行记录。

DB2、Oracle 和 SQL Server

本解决方案需要用到窗口函数。使用哪个窗口函数取决于我们希望如何处理 Tie[1]。下面的解决方案选择了 DENSE_RANK 函数，这意味着每一个 Tie 只会被计数一次。

注 1：此处 Tie 意为“平手、平局”。本书保持不译。在排序计算的过程中，如果一个名次上出现了多个候选项，则每一个候选项均可称之为“一个 Tie”。有的数据库函数的计算结果中允许出现多个 Tie，有的则仅返回一个。——译者注

```
1  select ename,sal
2    from (
3  select ename, sal,
4         dense_rank() over (order by sal desc) dr
5    from emp
6         ) x
7   where dr <= 5
```

上述查询返回的行数可能超过 5，但只有 5 种不同的工资值。如果你希望不考虑 Tie，只返回 5 行记录的话，那就使用 ROW_NUMBER OVER（因为该函数不关心 Tie）。

MySQL 和 PostgreSQL

使用标量子查询为每个工资值创建一个序号，然后通过上述序号限制子查询的结果行数。

```
 1  select ename,sal
 2    from (
 3  select (select count(distinct b.sal)
 4            from emp b
 5           where a.sal <= b.sal) as rnk,
 6          a.sal,
 7          a.ename
 8    from emp a
 9         )
10   where rnk <= 5
```

3. 讨论

DB2、Oracle 和 SQL Server

内嵌视图 X 里的窗口函数 DENSE_RANK OVER 完成了全部工作，执行该函数后得到的结果如下所示。

```
select ename, sal,
       dense_rank() over (order by sal desc) dr
  from emp

ENAME      SAL         DR
------- ------ ----------
KING      5000          1
SCOTT     3000          2
FORD      3000          2
JONES     2975          3
BLAKE     2850          4
CLARK     2450          5
ALLEN     1600          6
TURNER    1500          7
MILLER    1300          8
WARD      1250          9
MARTIN    1250          9
ADAMS     1100         10
JAMES      950         11
SMITH      800         12
```

最后，我们只需要返回 DR 小于或等于 5 的行即可。

MySQL 和 PostgreSQL

内嵌视图 X 里的标量子查询为工资值编排序号，结果如下。

```
select (select count(distinct b.sal)
          from emp b
         where a.sal <= b.sal) as rnk,
       a.sal,
       a.ename
  from emp a

RNK    SAL ENAME
--- ------ -------
  1   5000 KING
  2   3000 SCOTT
  2   3000 FORD
  3   2975 JONES
  4   2850 BLAKE
  5   2450 CLARK
  6   1600 ALLEN
  7   1500 TURNER
  8   1300 MILLER
  9   1250 WARD
  9   1250 MARTIN
 10   1100 ADAMS
 11    950 JAMES
 12    800 SMITH
```

最后，返回 RNK 小于或等于 5 的那些行即可。

11.6 找出最大和最小的记录

1. 问题

你想找出表里的“极端”值。例如，你希望找到 EMP 表中工资最高和最低的员工。

2. 解决方案

DB2、Oracle 和 SQL Server

使用窗口函数 MIN OVER 和 MAX OVER 分别找出最低和最高的工资值，内嵌视图的查询结果集如下所示。

```
1  select ename
2    from (
3  select ename, sal,
4         min(sal)over() min_sal,
5         max(sal)over() max_sal
6    from emp
7         ) x
8   where sal in (min_sal,max_sal)
```

MySQL 和 PostgreSQL

使用两个子查询，分别返回 SAL 的最小值和最大值。

```
1  select ename
2    from emp
3   where sal in ( (select min(sal) from emp),
4                  (select max(sal) from emp) )
```

3. 讨论

DB2、Oracle 和 SQL Server

有了窗口函数 MIN OVER 和 MAX OVER，我们就能在每一行里访问到最低和最高的工资值。内嵌视图 X 的查询结果集如下所示。

```
select ename, sal,
       min(sal)over() min_sal,
       max(sal)over() max_sal
  from emp

ENAME      SAL    MIN_SAL    MAX_SAL
------- ------ ---------- ----------
SMITH      800        800       5000
ALLEN     1600        800       5000
WARD      1250        800       5000
JONES     2975        800       5000
MARTIN    1250        800       5000
BLAKE     2850        800       5000
CLARK     2450        800       5000
SCOTT     3000        800       5000
KING      5000        800       5000
TURNER    1500        800       5000
ADAMS     1100        800       5000
JAMES      950        800       5000
FORD      3000        800       5000
MILLER    1300        800       5000
```

得到了上述结果集之后，剩下的工作就是返回 SAL 等于 MIN_SAL 或 MAX_SAL 的行。

MySQL 和 PostgreSQL

本解决方案在 IN 列表里用了两个子查询分别找出 EMP 表的最低和最高工资值。外层查询返回的行就是和子查询返回值相匹配的结果。

11.7 查询未来的行

1. 问题

如果有员工的工资低于紧随其后入职的同事，那么你希望把这些人找出来。我们先看一下如下所示的结果集。

```
ENAME             SAL HIREDATE
---------- ---------- ---------
SMITH             800 17-DEC-80
ALLEN            1600 20-FEB-81
WARD             1250 22-FEB-81
JONES            2975 02-APR-81
BLAKE            2850 01-MAY-81
```

```
CLARK          2450 09-JUN-81
URNER         1500 08-SEP-81
MARTIN         1250 28-SEP-81
KING           5000 17-NOV-81
JAMES           950 03-DEC-81
FORD           3000 03-DEC-81
MILLER         1300 23-JAN-82
SCOTT          3000 09-DEC-82
ADAMS          1100 12-JAN-83
```

SMITH、WARD、MARTIN、JAMES 以及 MILLER 的工资低于紧随其后入职的同事，因此，他们就是你要找的查询结果。

2. 解决方案

首先要确定“未来”的含义。我们必须对结果集排序以便能明确判断某一行记录含有的值是否“晚”于另一行。

DB2、MySQL、PostgreSQL 和 SQL Server

使用子查询为每一位员工计算出如下的值。

- 入职比他晚、且工资更高的员工当中最早入职的那个人的入职日期。
- 入职比他晚的员工当中最早入职的那个人的入职日期。

如果上述两个日期相等，那么这个人就是我们要找的。

```
 1 select ename, sal, hiredate
 2   from (
 3 select a.ename, a.sal, a.hiredate,
 4        (select min(hiredate) from emp b
 5          where b.hiredate > a.hiredate
 6            and b.sal      > a.sal ) as next_sal_grtr,
 7        (select min(hiredate) from emp b
 8          where b.hiredate > a.hiredate) as next_hire
 9   from emp a
10        ) x
11  where next_sal_grtr = next_hire
```

Oracle

可以使用窗口函数 LEAD OVER 访问下一个入职的员工的工资。剩下的就非常简单了，只需要检查一下该工资值是否更高即可。

```
1 select ename, sal, hiredate
2   from (
3 select ename, sal, hiredate,
4        lead(sal)over(order by hiredate) next_sal
5   from emp
6        )
7  where sal < next_sal
```

3. 讨论

DB2、MySQL、PostgreSQL 和 SQL Server

标量子查询为每一位员工返回紧随其后入职的第一个人的 HIREDATE，以及入职时间更晚且

工资更高的第一个人的 HIREDATE。下面展示了未经过滤处理的数据。

```
select a.ename, a.sal, a.hiredate,
       (select min(hiredate) from emp b
         where b.hiredate > a.hiredate
           and b.sal      > a.sal ) as next_sal_grtr,
       (select min(hiredate) from emp b
         where b.hiredate > a.hiredate) as next_hire
  from emp a

ENAME      SAL HIREDATE  NEXT_SAL_GRTR NEXT_HIRE
------- ------ --------- ------------- ---------
SMITH      800 17-DEC-80 20-FEB-81     20-FEB-81
ALLEN     1600 20-FEB-81 02-APR-81     22-FEB-81
WARD      1250 22-FEB-81 02-APR-81     02-APR-81
JONES     2975 02-APR-81 17-NOV-81     01-MAY-81
MARTIN    1250 28-SEP-81 17-NOV-81     17-NOV-81
BLAKE     2850 01-MAY-81 17-NOV-81     09-JUN-81
CLARK     2450 09-JUN-81 17-NOV-81     08-SEP-81
SCOTT     3000 09-DEC-82               12-JAN-83
KING      5000 17-NOV-81               03-DEC-81
TURNER    1500 08-SEP-81 17-NOV-81     28-SEP-81
ADAMS     1100 12-JAN-83
JAMES      950 03-DEC-81 23-JAN-82     23-JAN-82
FORD      3000 03-DEC-81               23-JAN-82
MILLER    1300 23-JAN-82 09-DEC-82     09-DEC-82
```

对于当前员工而言，入职时间比他晚、且工资更高的的人不一定就是紧随其后入职的第一人。下一步（也是最后一步）只返回 NEXT_SAL_GRTR（入职时间晚于当前员工、且工资更高的员工当中最早入职的那个人的 HIREDATE）等于 NEXT_HIRE（入职比他晚的员工当中最早入职的那个人的 HIREDATE）的行。

Oracle

窗口函数 LEAD OVER 正好可以解决这一类问题。LEAD OVER 不仅使得代码更具可读性，同时也让解决方案变得更灵活，因为我们可以传递一个参数给 LEAD OVER，告诉它需要往前看未来多少行的数据（默认值是 1 行）。在排好序的数据集里如果含有重复数据，那么这种情况下能够往前看到多于 1 行的数据是很重要的。

下面的例子展示了使用 LEAD OVER 来看“下一个”入职的员工的工资是多么方便。

```
select ename, sal, hiredate,
       lead(sal)over(order by hiredate) next_sal
  from emp

ENAME      SAL HIREDATE    NEXT_SAL
------- ------ --------- ----------
SMITH      800 17-DEC-80       1600
ALLEN     1600 20-FEB-81       1250
WARD      1250 22-FEB-81       2975
JONES     2975 02-APR-81       2850
BLAKE     2850 01-MAY-81       2450
CLARK     2450 09-JUN-81       1500
TURNER    1500 08-SEP-81       1250
```

```
MARTIN    1250 28-SEP-81       5000
KING      5000 17-NOV-81        950
JAMES      950 03-DEC-81       3000
FORD      3000 03-DEC-81       1300
MILLER    1300 23-JAN-82       3000
SCOTT     3000 09-DEC-82       1100
ADAMS     1100 12-JAN-83
```

最后，筛选出 SAL 小于 NEXT_SAL 的行。因为 LEAD OVER 默认往前看 1 行，如果 EMP 表里有重复数据，尤其是当同一天入职的员工多于一个人的情况下，入职日期相同的员工之间也会做 SAL 比较。这可能偏离了我们的预期。如果我们想要把每一个员工的 SAL 和下一个同事的做比较，并且明确要求屏蔽掉在同一天入职的其他员工，那么就需要用到下面的替代方案。

```
select ename, sal, hiredate
  from (
select ename, sal, hiredate,
       lead(sal,cnt-rn+1)over(order by hiredate) next_sal
  from (
select ename,sal,hiredate,
       count(*)over(partition by hiredate) cnt,
       row_number()over(partition by hiredate order by empno) rn
  from emp
       )
       )
 where sal < next_sal
```

上述做法的关键在于找出从当前行到它应该与之比较的行之间的距离。例如，如果有 5 个重复行，那么它的第一行就需要跳过 5 行数据才能找到正确的 LEAD OVER 行。对于具有重复 HIREDATE 的员工而言，那么 CNT 代表了他们的 HIREDATE 一共在多少行里出现过。RN 的值代表了 DEPTNO 等于 10 的每一个员工的序号。该序号的生成按照 HIREDATE 分区，因此只有那些含有重复 HIREDATE 的员工才可能有大于 1 的 RN 值。生成序号的时候先基于 EMPNO 做了排序（我们只是随便选了 EMPNO 做排序的基准）。现在我们已经知道了有多少个重复项，而且每个重复项都有一个序号，那么与下一个 HIREDATE 的距离就是重复项的总数减去当前的序号再加 1，即“CNT−RN+1”。

4. 参考资料

其他关于使用 LEAD OVER 处理重复项的例子（并且有更详尽、更彻底的技术细节讨论）参见 8.7 节和 10.2 节。

11.8　行值轮转

1. 问题

你想返回每个员工的姓名、工资，以及下一个最高和最低的工资值。如果没有找到更高或更低的工资值，你希望结果集可以“折回”（第一个 SAL 的前一行是最后一个 SAL；反之，最后一个 SAL 的下一行即是第一个 SAL）。你希望返回如下所示的结果集。

```
ENAME             SAL    FORWARD     REWIND
---------- ---------- ---------- ----------
SMITH             800        950       5000
JAMES             950       1100        800
ADAMS            1100       1250        950
WARD             1250       1250       1100
MARTIN           1250       1300       1250
MILLER           1300       1500       1250
TURNER           1500       1600       1300
ALLEN            1600       2450       1500
CLARK            2450       2850       1600
BLAKE            2850       2975       2450
JONES            2975       3000       2850
SCOTT            3000       3000       2975
FORD             3000       5000       3000
KING             5000        800       3000
```

2. 解决方案

对于 Oracle 用户而言，窗口函数 LEAD OVER 和 LAG OVER 使得本问题解决起来相对容易，而且代码可读性更好。对于其他数据库，可以使用标量子查询，不过 Tie 可能会带来问题。由于存在 Tie 的问题，对于不支持窗口函数的关系数据库管理系统，我们只能提供一个近似的解决方案。

DB2、SQL Server、MySQL 和 PostgreSQL

使用标量子查询为每一个工资值找到它的下一个和前一个的工资值。

```
 1  select e.ename, e.sal,
 2         coalesce(
 3           (select min(sal) from emp d where d.sal > e.sal),
 4           (select min(sal) from emp)
 5         ) as forward,
 6         coalesce(
 7           (select max(sal) from emp d where d.sal < e.sal),
 8           (select max(sal) from emp)
 9         ) as rewind
10    from emp e
11   order by 2
```

Oracle

使用窗口函数 LAG OVER 和 LEAD OVER 访问当前行的上一行和下一行记录。

```
1 select ename,sal,
2        nvl(lead(sal)over(order by sal),min(sal)over()) forward,
3        nvl(lag(sal)over(order by sal),max(sal)over()) rewind
4   from emp
```

3. 讨论

DB2、SQL Server、MySQL 和 PostgreSQL

标量子查询方案并没有真正解决本问题。它只是一个近似的方案，当两行记录包含相同的 SAL 时，该解决方案就会返回不正确的结果。不过，在没有窗口函数可用的情况下，它已经是最好的方案了。

Oracle

（默认情况下，除非有特别指定。）窗口函数 LAG OVER 和 LEAD OVER 将分别返回当前行的上一行和下一行记录。“上一行”或“下一行”取决于 OVER 子句里的 ORDER BY 部分。如果仔细阅读本解决方案的代码，我们会发现它首先按照 SAL 排序数据集，并提取出了当前行的上一行和下一行。

```
select ename,sal,
       lead(sal)over(order by sal) forward,
       lag(sal)over(order by sal) rewind
  from emp

ENAME             SAL    FORWARD     REWIND
---------- ---------- ---------- ----------
SMITH             800        950
JAMES             950       1100        800
ADAMS            1100       1250        950
WARD             1250       1250       1100
MARTIN           1250       1300       1250
MILLER           1300       1500       1250
TURNER           1500       1600       1300
ALLEN            1600       2450       1500
CLARK            2450       2850       1600
BLAKE            2850       2975       2450
JONES            2975       3000       2850
SCOTT            3000       3000       2975
FORD             3000       5000       3000
KING             5000                  3000
```

注意，员工 SMITH 的 REWIND 是 Null，而 KING 的 FORWARD 也是 Null；这是因为两个人的 SAL 分别是最低值和最高值。“问题”部分提到，FORWARD 或 REWIND 若出现 Null 值，则应该“折回”。这就意味着，对于最大的 SAL，FORWARD 值应为 EMP 表中最小的 SAL；而对于最小的 SAL，REWIND 值应为最大的 SAL。没有指定分区（即 OVER 子句后面跟一对空括号）的窗口函数 MIN OVER 和 MAX OVER 将分别返回最大和最小的 SAL。结果集如下所示。

```
select ename,sal,
       nvl(lead(sal)over(order by sal),min(sal)over()) forward,
       nvl(lag(sal)over(order by sal),max(sal)over()) rewind
  from emp

ENAME             SAL    FORWARD     REWIND
---------- ---------- ---------- ----------
SMITH             800        950       5000
JAMES             950       1100        800
ADAMS            1100       1250        950
WARD             1250       1250       1100
MARTIN           1250       1300       1250
MILLER           1300       1500       1250
TURNER           1500       1600       1300
ALLEN            1600       2450       1500
CLARK            2450       2850       1600
BLAKE            2850       2975       2450
JONES            2975       3000       2850
```

```
SCOTT            3000       3000       2975
FORD             3000       5000       3000
KING             5000        800       3000
```

LAG OVER 和 LEAD OVER 还有一个非常有用的功能，就是可以指定向前或者向后移动多少行。对于本例而言，我们只往前或往后移动了一行。如果你想往前移动 3 行，并且往后移动 5 行，做法非常简单。只需要指定移动值分别为 3 和 5 即可，如下所示。

```
select ename,sal,
       lead(sal,3)over(order by sal) forward,
       lag(sal,5)over(order by sal) rewind
  from emp

ENAME             SAL    FORWARD     REWIND
---------- ---------- ---------- ----------
SMITH             800       1250
JAMES             950       1250
ADAMS            1100       1300
WARD             1250       1500
MARTIN           1250       1600
MILLER           1300       2450        800
TURNER           1500       2850        950
ALLEN            1600       2975       1100
CLARK            2450       3000       1250
BLAKE            2850       3000       1250
JONES            2975       5000       1300
SCOTT            3000                  1500
FORD             3000                  1600
KING             5000                  2450
```

11.9　对结果排序

1. 问题

你想对 EMP 表里的工资值排序，并且允许 Tie。你希望返回如下所示的结果集。

```
RNK        SAL
--- ----------
  1        800
  2        950
  3       1100
  4       1250
  4       1250
  5       1300
  6       1500
  7       1600
  8       2450
  9       2850
 10       2975
 11       3000
 11       3000
 12       5000
```

2. 解决方案

窗口函数使得排序操作变得极其简单、方便。有 3 种窗口函数对于排序非常有用：DENSE_RANK OVER、ROW_NUMBER OVER 和 RANK OVER。

DB2、Oracle 和 SQL Server

因为允许 Tie，所以这里选择了窗口函数 DENSE_RANK OVER。

```
1 select dense_rank() over(order by sal) rnk, sal
2   from emp
```

MySQL 和 PostgreSQL

因为不支持窗口函数功能，我们应该用标量子查询来对工资排序。

```
1  select (select count(distinct b.sal)
2            from emp b
3           where b.sal <= a.sal) as rnk,
4         a.sal
5    from emp a
```

3. 讨论

DB2、Oracle 和 SQL Server

窗口函数 DENSE_RANK OVER 完成了大部分工作。OVER 关键字后面的括号里要放一个 ORDER BY 子句以指定用于排序的列。本解决方案使用的是 ORDER BY SAL，因此 EMP 表是按照工资递增的顺序来排序的。

MySQL 和 PostgreSQL

标量子查询的输出结果类似于 DENSE_RANK 函数，因为标量子查询是基于 SAL 展开迭代计算处理的。

11.10　删除重复项

1. 问题

你想找出 EMP 表里不同的职位种类，但又不希望看到重复项。结果集应该如下所示。

```
JOB
---------
ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN
```

2. 解决方案

所有的关系数据库管理系统都支持 DISTINCT 关键字，并且它也是从结果集中删除重复项的最简单的方法。然而，本实例将展示另外两种删除重复项的做法。

DB2、Oracle 和 SQL Server

传统的做法是使用 DISTINCT，有时候也会用 GROUP BY（正如后面的 MySQL 和 PostgreSQL 解决方案），这些方法适用于所有的关系数据库管理系统。下面给出的替代解决方案则使

用了窗口函数 ROW_NUMBER OVER。

```
1  select job
2    from (
3  select job,
4         row_number()over(partition by job order by job) rn
5    from emp
6         )x
7   where rn = 1
```

MySQL 和 PostgreSQL

使用 DISTINCT 关键字从结果集里删除重复项。

```
select distinct job
  from emp
```

另外，也可以通过 GROUP BY 达到同样目的。

```
select job
  from emp
 group by job
```

3. 讨论

DB2、Oracle 和 SQL Server

本解决方案展示了一种分区窗口函数（partitioned window function）的特殊用法。通过在 OVER 子句里使用 PARTITION BY，当一种新的职位出现时，ROW_NUMBER 的返回值会被重置为 1。内嵌视图 X 的结果集如下所示。

```
select job,
       row_number()over(partition by job order by job) rn
  from emp

JOB               RN
--------- ----------
ANALYST            1
ANALYST            2
CLERK              1
CLERK              2
CLERK              3
CLERK              4
MANAGER            1
MANAGER            2
MANAGER            3
PRESIDENT          1
SALESMAN           1
SALESMAN           2
SALESMAN           3
SALESMAN           4
```

每一行都被分配了一个递增的序号，当职位种类发生改变时，序号也会被重置为 1。为了删除重复项，我们需要筛选出 RN 等于 1 的那些行。

使用 ROW_NUMBER OVER 时，必须有一个 ORDER BY 子句（DB2 除外），但这并不影响最终结

果。我们只是希望提取出每一种职位，至于每一个职位来自哪一行其实无关紧要。

MySQL 和 PostgreSQL

第一种解决方案展示了如何使用关键字 DISTINCT 从结果集里删除重复项。需要注意的是，DISTINCT 会对整个 SELECT 列表做限制；增加一列或者几列会改变最终的结果集。考虑下面两个查询的差异。

```
select distinct job           select distinct job, deptno
  from emp                      from emp

JOB                             JOB           DEPTNO
---------                       --------- ----------
ANALYST                         ANALYST           20
CLERK                           CLERK             10
MANAGER                         CLERK             20
PRESIDENT                       CLERK             30
SALESMAN                        MANAGER           10
                                MANAGER           20
                                MANAGER           30
                                PRESIDENT         10
                                SALESMAN          30
```

向 SELECT 列表里增加 DEPTNO 列的话，返回的就变成了 EMP 表里不同的 JOB/DEPTNO 组合值。

第二种解决方案使用 GROUP BY 删除重复项，以这种方式使用 GROUP BY 并不罕见。注意，GROUP BY 和 DISTINCT 是两个非常不同的子句，它们是不可互换的。为了给出全部可行的解决方案，我特地把使用 GROUP BY 去掉重复项的做法也包含进了本实例，这样读者下次看到类似做法时就不会有什么疑问了。

11.11　查找骑士值

1. 问题

你想返回一个结果集，其中包括每个员工的姓名、部门、工资、入职时间以及每一个部门里最近入职的那个员工的工资。你希望返回如下所示的结果集。

```
DEPTNO ENAME             SAL HIREDATE    LATEST_SAL
------ ---------- ---------- ----------- ----------
    10 MILLER           1300 23-JAN-1982       1300
    10 KING             5000 17-NOV-1981       1300
    10 CLARK            2450 09-JUN-1981       1300
    20 ADAMS            1100 12-JAN-1983       1100
    20 SCOTT            3000 09-DEC-1982       1100
    20 FORD             3000 03-DEC-1981       1100
    20 JONES            2975 02-APR-1981       1100
    20 SMITH             800 17-DEC-1980       1100
    30 JAMES             950 03-DEC-1981        950
    30 MARTIN           1250 28-SEP-1981        950
    30 TURNER           1500 08-SEP-1981        950
    30 BLAKE            2850 01-MAY-1981        950
    30 WARD             1250 22-FEB-1981        950
    30 ALLEN            1600 20-FEB-1981        950
```

我把 LATEST_SAL 值称作“骑士值”（knight value），因为找出它们的方法类似于国际象棋游戏中骑士的路径。我们找到结果的方式就像一个骑士确定新位置：跳到一行，然后转向并跳到一个不同的列（图 11-1）。为找到正确的 LATEST_SAL 值，你必须先定位（跳到）每个 DEPTNO 对应的最新 HIREDATE，然后选择（跳到）那一行的 SAL 列。

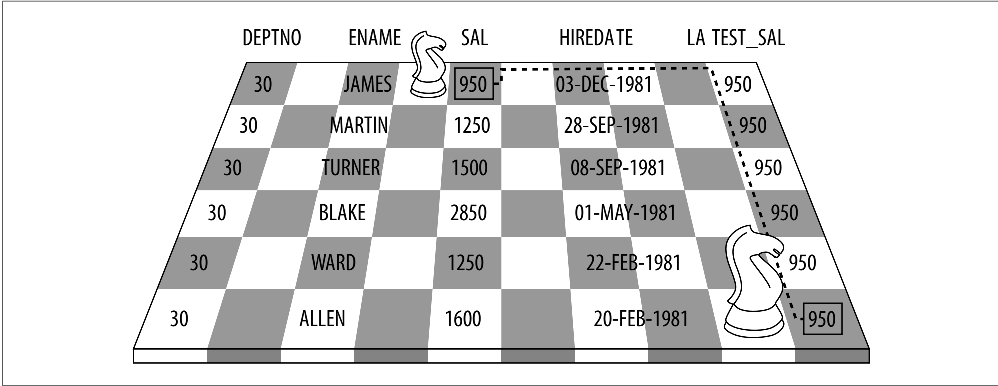


图 11-1：骑士值来源于国际象棋中骑士的走法



“骑士值”是由我的一个非常聪明的同事 Kay Young 创造出来的。在他帮我做完正确性检查后，我告诉他我被难住了，因为无法为本实例想出一个恰到好处的标题。考虑到需要先对当前行数据做出评估，然后“跳”到另外一行并取出某列的值，于是他提出了“骑士值”这个名词。

2. 解决方案

DB2 和 SQL Server

在子查询中使用 CASE 表达式，并返回每个 DEPTNO 对应的最近入职的那个员工的 SAL；对于其他工资值，则返回 0。在外层的查询中使用窗口函数 MAX OVER 为每个员工的部门返回非零的 SAL。

```
 1 select deptno,
 2        ename,
 3        sal,
 4        hiredate,
 5        max(latest_sal)over(partition by deptno) latest_sal
 6   from (
 7 select deptno,
 8        ename,
 9        sal,
10        hiredate,
11        case
12          when hiredate = max(hiredate)over(partition by deptno)
13          then sal else 0
14        end latest_sal
15   from emp
16        ) x
17  order by 1, 4 desc
```

MySQL 和 PostgreSQL

使用两层嵌套的标量子查询。首先，找出每个 DEPTO 对应的最近入职的员工的 HIREDATE。然后使用聚合函数 MAX（如果有重复项）找出每个 DEPTNO 对应的最近入职的员工的 SAL。

```
1  select e.deptno,
2         e.ename,
3         e.sal,
4         e.hiredate,
5         (select max(d.sal)
6            from emp d
7           where d.deptno  = e.deptno
8             and d.hiredate =
9                 (select max(f.hiredate)
10                   from emp f
11                  where f.deptno = e.deptno)) as latest_sal
12   from emp e
13  order by 1, 4 desc
```

Oracle

使用窗口函数 MAX OVER 返回每个 DEPTNO 对应的最高 SAL 值。在 KEEP 子句中使用函数 DENSE_RANK 和 LAST，并按照 HIREDATE 排序，为给定 DEPTNO 对应的最新 HIREDATE 返回最高的 SAL 值。

```
1  select deptno,
2         ename,
3         sal,
4         hiredate,
5         max(sal)
6           keep(dense_rank last order by hiredate)
7           over(partition by deptno) latest_sal
8    from emp
9   order by 1, 4 desc
```

3. 讨论

DB2 和 SQL Server

首先在 CASE 表达式中使用窗口函数 MAX OVER 找出每个 DEPTNO 对应的最近入职的员工。如果员工的 HIREDATE 等于 MAX OVER 的返回值，那么 CASE 表达式就会返回该员工的 SAL 值；否则，返回 0。这一步的结果如下所示。

```
select deptno,
      ename,
      sal,
      hiredate,
      case
         when hiredate = max(hiredate)over(partition by deptno)
         then sal else 0
      end latest_sal
  from emp

DEPTNO ENAME             SAL HIREDATE    LATEST_SAL
------ ---------- ---------- ----------- ----------
    10 CLARK            2450 09-JUN-1981          0
    10 KING             5000 17-NOV-1981          0
```

```
    10 MILLER           1300 23-JAN-1982       1300
    20 SMITH             800 17-DEC-1980          0
    20 ADAMS            1100 12-JAN-1983       1100
    20 FORD             3000 03-DEC-1981          0
    20 SCOTT            3000 09-DEC-1982          0
    20 JONES            2975 02-APR-1981          0
    30 ALLEN            1600 20-FEB-1981          0
    30 BLAKE            2850 01-MAY-1981          0
    30 MARTIN           1250 28-SEP-1981          0
    30 JAMES             950 03-DEC-1981        950
    30 TURNER           1500 08-SEP-1981          0
    30 WARD             1250 22-FEB-1981          0
```

LATEST_SAL 值要么是 0，要么是最近入职的员工的 SAL，因此我们可以把上述查询作为一个内嵌视图，并在此基础上再次使用 MAX OVER，这次我们要为每个 DEPTNO 返回最大的非零 LATEST_SAL。

```
select deptno,
       ename,
       sal,
       hiredate,
       max(latest_sal)over(partition by deptno) latest_sal
  from (
select deptno,
       ename,
       sal,
       hiredate,
       case
           when hiredate = max(hiredate)over(partition by deptno)
           then sal else 0
       end latest_sal
  from emp
       )x
 order by 1, 4 desc

DEPTNO ENAME              SAL HIREDATE    LATEST_SAL
------ ---------- ---------- ----------- ----------
    10 MILLER           1300 23-JAN-1982       1300
    10 KING             5000 17-NOV-1981       1300
    10 CLARK            2450 09-JUN-1981       1300
    20 ADAMS            1100 12-JAN-1983       1100
    20 SCOTT            3000 09-DEC-1982       1100
    20 FORD             3000 03-DEC-1981       1100
    20 JONES            2975 02-APR-1981       1100
    20 SMITH             800 17-DEC-1980       1100
    30 JAMES             950 03-DEC-1981        950
    30 MARTIN           1250 28-SEP-1981        950
    30 TURNER           1500 08-SEP-1981        950
    30 BLAKE            2850 01-MAY-1981        950
    30 WARD             1250 22-FEB-1981        950
    30 ALLEN            1600 20-FEB-1981        950
```

MySQL 和 PostgreSQL

首先，使用标量子查询找出每个 DEPTNO 对应的最新入职的员工的 HIREDATE。

```
select e.deptno,
       e.ename,
       e.sal,
       e.hiredate,
       (select max(f.hiredate)
          from emp f
         where f.deptno = e.deptno) as last_hire
  from emp e
 order by 1, 4 desc

DEPTNO ENAME             SAL HIREDATE    LAST_HIRE
------ ---------- ---------- ----------- -----------
    10 MILLER           1300 23-JAN-1982 23-JAN-1982
    10 KING             5000 17-NOV-1981 23-JAN-1982
    10 CLARK            2450 09-JUN-1981 23-JAN-1982
    20 ADAMS            1100 12-JAN-1983 12-JAN-1983
    20 SCOTT            3000 09-DEC-1982 12-JAN-1983
    20 FORD             3000 03-DEC-1981 12-JAN-1983
    20 JONES            2975 02-APR-1981 12-JAN-1983
    20 SMITH             800 17-DEC-1980 12-JAN-1983
    30 JAMES             950 03-DEC-1981 03-DEC-1981
    30 MARTIN           1250 28-SEP-1981 03-DEC-1981
    30 TURNER           1500 08-SEP-1981 03-DEC-1981
    30 BLAKE            2850 01-MAY-1981 03-DEC-1981
    30 WARD             1250 22-FEB-1981 03-DEC-1981
    30 ALLEN            1600 20-FEB-1981 03-DEC-1981
```

然后，找出每个 DEPTNO 对应的入职日期等于 LAST_HIRE 的员工的 SAL。使用聚合函数 MAX 找出最大值（如果同一天入职的员工不止一人的话）。

```
select e.deptno,
       e.ename,
       e.sal,
       e.hiredate,
       (select max(d.sal)
          from emp d
         where d.deptno   = e.deptno
           and d.hiredate =
               (select max(f.hiredate)
                  from emp f
                 where f.deptno = e.deptno)) as latest_sal
  from emp e
 order by 1, 4 desc

DEPTNO ENAME             SAL HIREDATE    LATEST_SAL
------ ---------- ---------- ----------- ----------
    10 MILLER           1300 23-JAN-1982       1300
    10 KING             5000 17-NOV-1981       1300
    10 CLARK            2450 09-JUN-1981       1300
    20 ADAMS            1100 12-JAN-1983       1100
    20 SCOTT            3000 09-DEC-1982       1100
    20 FORD             3000 03-DEC-1981       1100
    20 JONES            2975 02-APR-1981       1100
    20 SMITH             800 17-DEC-1980       1100
    30 JAMES             950 03-DEC-1981        950
    30 MARTIN           1250 28-SEP-1981        950
```

```
   30 TURNER           1500 08-SEP-1981        950
   30 BLAKE            2850 01-MAY-1981        950
   30 WARD             1250 22-FEB-1981        950
   30 ALLEN            1600 20-FEB-1981        950
```

Oracle

Oracle 8*i* 用户不妨采用前面的 DB2 解决方案。Oracle 9*i* 及后续版本则可以使用下面给出的解决方案。如下 Oracle 解决方案的关键在于利用 KEEP 子句。KEEP 子句为分组或者分区之后的行数据进行排序，并取出每组的第一行或最后一行。试想一下，去掉了 KEEP 子句的解决方案会如何。

```
select deptno,
       ename,
       sal,
       hiredate,
       max(sal) over(partition by deptno) latest_sal
  from emp
 order by 1, 4 desc

DEPTNO ENAME             SAL HIREDATE    LATEST_SAL
------ ---------- ---------- ----------- ----------
    10 MILLER           1300 23-JAN-1982       5000
    10 KING             5000 17-NOV-1981       5000
    10 CLARK            2450 09-JUN-1981       5000
    20 ADAMS            1100 12-JAN-1983       3000
    20 SCOTT            3000 09-DEC-1982       3000
    20 FORD             3000 03-DEC-1981       3000
    20 JONES            2975 02-APR-1981       3000
    20 SMITH             800 17-DEC-1980       3000
    30 JAMES             950 03-DEC-1981       2850
    30 MARTIN           1250 28-SEP-1981       2850
    30 TURNER           1500 08-SEP-1981       2850
    30 BLAKE            2850 01-MAY-1981       2850
    30 WARD             1250 22-FEB-1981       2850
    30 ALLEN            1600 20-FEB-1981       2850
```

去掉了 KEEP 子句的 MAX OVER 只会简单地返回每个 DEPTNO 对应的最高工资值，而不是最近入职的那个员工的 SAL。这里的 KEEP 子句通过指定 ORDER BY HIREDATE 按照 HIREDATE 为每个 DEPTNO 对应的工资值进行排序。然后，DENSE_RANK 函数按照升序为 HIREDATE 排序。最后，函数 LAST 决定针对哪一行记录使用聚合函数：基于 DENSE_RANK 排序最后的一行。对本例而言，聚合函数 MAX 针对的是最后一行 HIREDATE 所对应的 SAL 列。其实就是为了保留每个 DEPTNO 对应的 HIREDATE 排序最后的那个 SAL 值。

我们基于某一列（HIREDATE）为每个 DEPTNO 对应的行做排序，却针对另一列（SAL）做聚合计算（MAX）。Oracle 具备的这种先针对某一个维度做排序，继而又针对另一个维度做聚合计算的能力非常有用，其他数据库需要额外的连接查询和内嵌视图才能达到同样的效果。最后，通过在 KEEP 子句后面跟一个 OVER 子句，我们就能返回由 KEEP 子句为每一行“保留”下来的 SAL 值。

另外，我们还可以对 HIREDATE 实行降序排列，并保留第一个 SAL 值。比较下面的两个查

询，它们都返回相同的结果集。

```
select deptno,
       ename,
       sal,
       hiredate,
       max(sal)
         keep(dense_rank last order by hiredate)
         over(partition by deptno) latest_sal
  from emp
 order by 1, 4 desc

DEPTNO ENAME             SAL HIREDATE    LATEST_SAL
------ ---------- ---------- ----------- ----------
    10 MILLER           1300 23-JAN-1982       1300
    10 KING             5000 17-NOV-1981       1300
    10 CLARK            2450 09-JUN-1981       1300
    20 ADAMS            1100 12-JAN-1983       1100
    20 SCOTT            3000 09-DEC-1982       1100
    20 FORD             3000 03-DEC-1981       1100
    20 JONES            2975 02-APR-1981       1100
    20 SMITH             800 17-DEC-1980       1100
    30 JAMES             950 03-DEC-1981        950
    30 MARTIN           1250 28-SEP-1981        950
    30 TURNER           1500 08-SEP-1981        950
    30 BLAKE            2850 01-MAY-1981        950
    30 WARD             1250 22-FEB-1981        950
    30 ALLEN            1600 20-FEB-1981        950


select deptno,
       ename,
       sal,
       hiredate,
       max(sal)
         keep(dense_rank first order by hiredate desc)
         over(partition by deptno) latest_sal
  from emp
 order by 1, 4 desc

DEPTNO ENAME             SAL HIREDATE    LATEST_SAL
------ ---------- ---------- ----------- ----------
    10 MILLER           1300 23-JAN-1982       1300
    10 KING             5000 17-NOV-1981       1300
    10 CLARK            2450 09-JUN-1981       1300
    20 ADAMS            1100 12-JAN-1983       1100
    20 SCOTT            3000 09-DEC-1982       1100
    20 FORD             3000 03-DEC-1981       1100
    20 JONES            2975 02-APR-1981       1100
    20 SMITH             800 17-DEC-1980       1100
    30 JAMES             950 03-DEC-1981        950
    30 MARTIN           1250 28-SEP-1981        950
    30 TURNER           1500 08-SEP-1981        950
    30 BLAKE            2850 01-MAY-1981        950
    30 WARD             1250 22-FEB-1981        950
    30 ALLEN            1600 20-FEB-1981        950
```

11.12　生成简单的预测

1. 问题

基于当前数据，你想得到代表未来行为的、额外的行和列。例如，考虑下面的结果集。

```
ID ORDER_DATE  PROCESS_DATE
-- ----------- ------------
 1 25-SEP-2005 27-SEP-2005
 2 26-SEP-2005 28-SEP-2005
 3 27-SEP-2005 29-SEP-2005
```

对于上述结果集里的每一行，你希望返回三行数据（对每一份订单，在现有数据行的基础上额外加上两行数据）。此外，在这些额外的行里，你希望增加两列以展示预期的订单处理日期。

根据上面提供的结果集，我们可以看到一个订单的处理需要花费两天时间。假设订单处理完成后，下一步要做订单核查，最后一步则是发货。核查发生在订单处理后一天，发货则是在核查后一天。你希望得到一个能展示整个过程的结果集。最终是要把上面的结果集变换为如下所示的结果集。

```
ID ORDER_DATE  PROCESS_DATE VERIFIED    SHIPPED
-- ----------- ------------ ----------- -----------
 1 25-SEP-2005 27-SEP-2005
 1 25-SEP-2005 27-SEP-2005  28-SEP-2005
 1 25-SEP-2005 27-SEP-2005  28-SEP-2005 29-SEP-2005
 2 26-SEP-2005 28-SEP-2005
 2 26-SEP-2005 28-SEP-2005  29-SEP-2005
 2 26-SEP-2005 28-SEP-2005  29-SEP-2005 30-SEP-2005
 3 27-SEP-2005 29-SEP-2005
 3 27-SEP-2005 29-SEP-2005  30-SEP-2005
 3 27-SEP-2005 29-SEP-2005  30-SEP-2005 01-OCT-2005
```

2. 解决方案

关键在于借助笛卡儿积为每一个订单生成额外的两行数据，然后只需使用 `CASE` 表达式生成所需的列值即可。

DB2 和 SQL Server

使用 `WITH` 递归查询为笛卡儿积产生所需数目的行。DB2 和 SQL Server 的解决方案几乎完全一样，除了获取当前日期的函数不同。DB2 调用 `CURRENT_DATE` 函数，而 SQL Server 使用 `GETDATE` 函数。下面展示了 SQL Server 的解决方案。

```
 1 with nrows(n) as (
 2 select 1 from t1 union all
 3 select n+1 from nrows where n+1 <= 3
 4 )
 5 select id,
 6        order_date,
 7        process_date,
 8        case when nrows.n >= 2
 9             then process_date+1
10             else null
```

```
11        end as verified,
12        case when nrows.n = 3
13             then process_date+2
14             else null
15        end  as shipped
16   from (
17 select nrows.n id,
18        getdate()+nrows.n   as order_date,
19        getdate()+nrows.n+2 as process_date
20   from nrows
21        ) orders, nrows
22  order by 1
```

Oracle

使用 CONNECT BY 子句生成笛卡儿积所需的 3 行数据。使用 WITH 子句，这样我们就可以重新使用由 CONNECT BY 返回的结果集，而不必再次调用它。

```
 1  with nrows as (
 2  select level n
 3    from dual
 4 connect by level <= 3
 5  )
 6  select id,
 7         order_date,
 8         process_date,
 9         case when nrows.n >= 2
10              then process_date+1
11              else null
12         end as verified,
13         case when nrows.n = 3
14              then process_date+2
15              else null
16         end  as shipped
17    from (
18  select nrows.n id,
19         sysdate+nrows.n   as order_date,
20         sysdate+nrows.n+2 as process_date
21    from nrows
22         ) orders, nrows
```

PostgreSQL

可以通过很多种不同的方式创建笛卡儿积，本解决方案用到了 PostgreSQL 的 GENERATE_SERIES 函数。

```
 1  select id,
 2         order_date,
 3         process_date,
 4         case when gs.n >= 2
 5              then process_date+1
 6              else null
 7         end as verified,
 8         case when gs.n = 3
 9              then process_date+2
10              else null
```

```
11          end  as shipped
12     from (
13   select gs.id,
14          current_date+gs.id   as order_date,
15          current_date+gs.id+2 as process_date
16     from generate_series(1,3) gs (id)
17          ) orders,
18            generate_series(1,3)gs(n)
```

MySQL

MySQL 不支持自动生成行数据的函数。

3. 讨论

DB2 和 SQL Server

“问题”部分提供的结果集来自内嵌视图 ORDERS，如下所示。

```
with nrows(n) as (
select 1 from t1 union all
select n+1 from nrows where n+1 <= 3
)
select nrows.n id,
       getdate()+nrows.n   as order_date,
       getdate()+nrows.n+2 as process_date
  from nrows


ID ORDER_DATE  PROCESS_DATE
-- ----------- ------------
 1 25-SEP-2005 27-SEP-2005
 2 26-SEP-2005 28-SEP-2005
 3 27-SEP-2005 29-SEP-2005
```

上面的查询使用 WITH 子句生成了 3 行数据，代表了我们要处理的订单。NROWS 返回了 3 个数字：1、2、3，这些数字和 GETDATE（DB2 则是 CURRENT_DATE）相加就得到了这些订单的 ORDER_DATE。因为前面在“问题”部分里说过订单处理需要花费 2 天时间，于是上面的查询也在 ORDER_DATE 的基础上增加了 2 天（将 NROWS 的返回值加入 GETDATE，然后再加上 2 天）。

现在我们已经得到了基础结果集，下一步是产生一个笛卡儿积，因为需要为每个订单返回 3 行数据。使用 NROWS 产生笛卡儿积，为每个订单返回 3 行数据。

```
with nrows(n) as (
select 1 from t1 union all
select n+1 from nrows where n+1 <= 3
)
select nrows.n,
       orders.*
  from (
select nrows.n id,
       getdate()+nrows.n   as order_date,
       getdate()+nrows.n+2 as process_date
  from nrows
       ) orders, nrows
```

```
 order by 2,1


  N  ID ORDER_DATE  PROCESS_DATE
--- --- ----------- ------------
  1   1 25-SEP-2005 27-SEP-2005
  2   1 25-SEP-2005 27-SEP-2005
  3   1 25-SEP-2005 27-SEP-2005
  1   2 26-SEP-2005 28-SEP-2005
  2   2 26-SEP-2005 28-SEP-2005
  3   2 26-SEP-2005 28-SEP-2005
  1   3 27-SEP-2005 29-SEP-2005
  2   3 27-SEP-2005 29-SEP-2005
  3   3 27-SEP-2005 29-SEP-2005
```

现在每个订单都有了 3 行记录，然后只要用 CASE 表达式生成额外的列值即可，这些列代表了订单核查和发货的状态。

每个订单的第一行记录里 VERIFIED 和 SHIPPED 应该是 Null，第二行的 SHIPPED 应该为 Null。第三行的 VERIFIED 和 SHIPPED 都应该是非 Null 值。最终的结果集如下所示。

```
with nrows(n) as (
select 1 from t1 union all
select n+1 from nrows where n+1 <= 3
)
select id,
       order_date,
       process_date,
       case when nrows.n >= 2
            then process_date+1
            else null
       end as verified,
       case when nrows.n = 3
            then process_date+2
            else null
       end  as shipped
  from (
select nrows.n id,
       getdate()+nrows.n   as order_date,
       getdate()+nrows.n+2 as process_date
  from nrows
       ) orders, nrows
 order by 1


ID ORDER_DATE  PROCESS_DATE VERIFIED    SHIPPED
-- ----------- ----------- ----------- -----------
 1 25-SEP-2005 27-SEP-2005
 1 25-SEP-2005 27-SEP-2005  28-SEP-2005
 1 25-SEP-2005 27-SEP-2005  28-SEP-2005 29-SEP-2005
 2 26-SEP-2005 28-SEP-2005
 2 26-SEP-2005 28-SEP-2005  29-SEP-2005
 2 26-SEP-2005 28-SEP-2005  29-SEP-2005 30-SEP-2005
 3 27-SEP-2005 29-SEP-2005
```

```
 3 27-SEP-2005 29-SEP-2005  30-SEP-2005
 3 27-SEP-2005 29-SEP-2005  30-SEP-2005 01-OCT-2005
```

最终的结果集展示了从收到订单直至发货为止的订单处理全过程。

Oracle

“问题”部分里的结果集是由内嵌视图 ORDERS 生成的，如下所示。

```
with nrows as (
select level n
  from dual
connect by level <= 3
)
select nrows.n id,
       sysdate+nrows.n   order_date,
       sysdate+nrows.n+2 process_date
  from nrows


ID ORDER_DATE  PROCESS_DATE
-- ----------- ------------
 1 25-SEP-2005 27-SEP-2005
 2 26-SEP-2005 28-SEP-2005
 3 27-SEP-2005 29-SEP-2005
```

上述查询借助 CONNECT BY 生成我们要处理的 3 行订单数据。 使用 WITH 子句参照 CONNECT BY 返回的行数据 NROWS.N。CONNECT BY 返回了数字 1、2 和 3，这些数字分别加上 SYSDATE 后得到的结果即是 ORDER_DATE。“问题”部分里提到，订单处理需要两天时间，因此上面的查询也为 ORDER_DATE 加上了 2 天（把 CONNECT BY 的返回值和 SYSDATE 相加后，再加上两天）。

现在得到了基础结果集，下一步是产生一个笛卡儿积，因为本问题要求为每个订单返回 3 行数据。使用 NROWS 产生笛卡儿积并为每个订单返回 3 行记录。

```
with nrows as (
select level n
  from dual
connect by level <= 3
)
select nrows.n,
       orders.*
  from (
select nrows.n id,
       sysdate+nrows.n   order_date,
       sysdate+nrows.n+2 process_date
  from nrows
       ) orders, nrows


  N  ID ORDER_DATE  PROCESS_DATE
--- --- ----------- ------------
  1   1 25-SEP-2005 27-SEP-2005
  2   1 25-SEP-2005 27-SEP-2005
```

```
  3   1 25-SEP-2005 27-SEP-2005
  1   2 26-SEP-2005 28-SEP-2005
  2   2 26-SEP-2005 28-SEP-2005
  3   2 26-SEP-2005 28-SEP-2005
  1   3 27-SEP-2005 29-SEP-2005
  2   3 27-SEP-2005 29-SEP-2005
  3   3 27-SEP-2005 29-SEP-2005
```

现在每个订单都有了 3 行数据，接下来只要使用 CASE 表达式生成额外的列值即可，这些列代表了订单核查和发货的状态。

每个订单的第一行记录里 VERIFIED 和 SHIPPED 应该是 Null。第二行的 SHIPPED 应该为 Null。第三行的 VERIFIED 和 SHIPPED 都应该是非 Null 值。最终的结果集如下所示。

```
with nrows as (
select level n
  from dual
connect by level <= 3
)
select id,
       order_date,
       process_date,
       case when nrows.n >= 2
            then process_date+1
            else null
       end as verified,
       case when nrows.n = 3
            then process_date+2
            else null
       end  as shipped
  from (
select nrows.n id,
       sysdate+nrows.n   order_date,
       sysdate+nrows.n+2 process_date
  from nrows
       ) orders, nrows


ID ORDER_DATE  PROCESS_DATE VERIFIED    SHIPPED
-- ----------- ----------- ----------- -----------
 1 25-SEP-2005 27-SEP-2005
 1 25-SEP-2005 27-SEP-2005  28-SEP-2005
 1 25-SEP-2005 27-SEP-2005  28-SEP-2005 29-SEP-2005
 2 26-SEP-2005 28-SEP-2005
 2 26-SEP-2005 28-SEP-2005  29-SEP-2005
 2 26-SEP-2005 28-SEP-2005  29-SEP-2005 30-SEP-2005
 3 27-SEP-2005 29-SEP-2005
 3 27-SEP-2005 29-SEP-2005  30-SEP-2005
 3 27-SEP-2005 29-SEP-2005  30-SEP-2005 01-OCT-2005
```

最终的结果集展示了从收到订单直至发货为止的订单处理全过程。

PostgreSQL

“问题”部分里的结果集是由内嵌视图 ORDERS 生成的，如下所示。

```
select gs.id,
       current_date+gs.id   as order_date,
       current_date+gs.id+2 as process_date
  from generate_series(1,3) gs (id)


ID ORDER_DATE  PROCESS_DATE
-- ----------- ------------
 1 25-SEP-2005 27-SEP-2005
 2 26-SEP-2005 28-SEP-2005
 3 27-SEP-2005 29-SEP-2005
```

上述查询使用 GENERATE_SERIES 函数生成了我们要处理的 3 行订单数据。GENERATE_SERIES 返回了数字 1、2 和 3，并把这些数字和 CURRENT_DATE 相加得到 ORDER_DATE。“问题”部分里提到，订单处理需要 2 天时间，因此上面的查询也为 ORDER_DATE 加上了 2 天（把 GENERATE_ SERIES 的返回值和 CURRENT_DATE 相加，然后再加上两天）。

现在我们得到了基础结果集，下一步是产生一个笛卡儿积，因为本问题要求为每个订单返回 3 行数据。使用 GENERATE_ SERIES 函数产生一个笛卡儿积并为每个订单返回 3 行记录。

```
select gs.n,
       orders.*
  from (
select gs.id,
       current_date+gs.id   as order_date,
       current_date+gs.id+2 as process_date
  from generate_series(1,3) gs (id)
       ) orders,
         generate_series(1,3)gs(n)


  N  ID ORDER_DATE  PROCESS_DATE
--- --- ----------- ------------
  1   1 25-SEP-2005 27-SEP-2005
  2   1 25-SEP-2005 27-SEP-2005
  3   1 25-SEP-2005 27-SEP-2005
  1   2 26-SEP-2005 28-SEP-2005
  2   2 26-SEP-2005 28-SEP-2005
  3   2 26-SEP-2005 28-SEP-2005
  1   3 27-SEP-2005 29-SEP-2005
  2   3 27-SEP-2005 29-SEP-2005
  3   3 27-SEP-2005 29-SEP-2005
```

现在每个顺序都拥有了 3 行，然后简单地用一个 CASE 表达式创建需要增加的列值，这些列代表了核查和装运的状态。

每个订单的第一行记录里 VERIFIED 和 SHIPPED 应该是 Null。第二行的 SHIPPED 应该为 Null。第三行的 VERIFIED 和 SHIPPED 都应该是非空值。最终的结果集如下所示。

```
select id,
       order_date,
       process_date,
       case when gs.n >= 2
```

```
            then process_date+1
            else null
        end as verified,
        case when gs.n = 3
             then process_date+2
             else null
        end  as shipped
  from (
select gs.id,
       current_date+gs.id   as order_date,
       current_date+gs.id+2 as process_date
  from generate_series(1,3) gs(id)
       ) orders,
         generate_series(1,3)gs(n)


ID ORDER_DATE  PROCESS_DATE VERIFIED    SHIPPED
-- ----------- ------------ ----------- -----------
 1 25-SEP-2005 27-SEP-2005
 1 25-SEP-2005 27-SEP-2005  28-SEP-2005
 1 25-SEP-2005 27-SEP-2005  28-SEP-2005 29-SEP-2005
 2 26-SEP-2005 28-SEP-2005
 2 26-SEP-2005 28-SEP-2005  29-SEP-2005
 2 26-SEP-2005 28-SEP-2005  29-SEP-2005 30-SEP-2005
 3 27-SEP-2005 29-SEP-2005
 3 27-SEP-2005 29-SEP-2005  30-SEP-2005
 3 27-SEP-2005 29-SEP-2005  30-SEP-2005 01-OCT-2005
```

最终的结果集展示了从收到订单直至发货为止的订单处理全过程。

第 12 章

报表和数据仓库

本章介绍了一些创建报表的实例。这些实例多数都与报表格式化和不同级别的聚合运算相关。本章另一部分重要的内容是结果集变换，例如，把行形式的数据转换为列形式。很多时候，结果集变换都非常有用。你若能深入理解并熟练掌握该技巧，必能在实际工作中举一反三，发现其更多的用武之地。

12.1 变换结果集成一行

1. 问题

你想把若干行数据重新组合成一行新数据，原始数据集的每一行变换后会作为新数据的一列出现。例如，下面的结果集展示了每个部门的员工人数。

```
DEPTNO        CNT
------ ----------
    10          3
    20          5
    30          6
```

你希望把上述结果集重新格式化成下面的输出结果。

```
DEPTNO_10  DEPTNO_20  DEPTNO_30
--------- ---------- ----------
        3          5          6
```

2. 解决方案

使用 CASE 表达式和聚合函数 SUM 实现结果集变换。

```
1  select sum(case when deptno=10 then 1 else 0 end) as deptno_10,
2         sum(case when deptno=20 then 1 else 0 end) as deptno_20,
```

```
3         sum(case when deptno=30 then 1 else 0 end) as deptno_30
4    from emp
```

3. 讨论

这是一个非常好的结果集变换入门实例。它的做法其实很简单：对于每一行的原始数据，使用 CASE 表达式把行变换成列。然后，由于本实例要合计每个部门的员工人数，因此需要调用聚合函数 SUM 计算出每个 DEPTNO 出现的次数。如果你还是不太理解以上解决方案的工作原理，那么不妨先执行下面的查询，该查询调用聚合函数 SUM，并加上了 DEPTNO 以增强可读性。

```
select deptno,
       case when deptno=10 then 1 else 0 end as deptno_10,
       case when deptno=20 then 1 else 0 end as deptno_20,
       case when deptno=30 then 1 else 0 end as deptno_30
  from emp
 order by 1

DEPTNO  DEPTNO_10  DEPTNO_20  DEPTNO_30
------ ---------- ---------- ----------
    10          1          0          0
    10          1          0          0
    10          1          0          0
    20          0          1          0
    20          0          1          0
    20          0          1          0
    20          0          1          0
    20          0          1          0
    30          0          0          1
    30          0          0          1
    30          0          0          1
    30          0          0          1
    30          0          0          1
    30          0          0          1
```

我们可以把每一个 CASE 表达式的结果值想象成一个标志位，它表示每个 DEPTNO 属于哪一行。至此，“把行变成列”的操作已经完成。下一步只要合计 DEPTNO_10、DEPTNO_20 和 DEPTNO_30 的值，并按照 DEPTNO 分组即可，结果集显示如下。

```
select deptno,
       sum(case when deptno=10 then 1 else 0 end) as deptno_10,
       sum(case when deptno=20 then 1 else 0 end) as deptno_20,
       sum(case when deptno=30 then 1 else 0 end) as deptno_30
  from emp
 group by deptno

DEPTNO  DEPTNO_10  DEPTNO_20  DEPTNO_30
------ ---------- ---------- ----------
    10          3          0          0
    20          0          5          0
    30          0          0          6
```

仔细观察上述结果集的话，可以发现上述输出结果在逻辑上是正确的。例如，DEPTNO 等于

10 的部门对应的行里面，DEPTNO_10 的值为 3，而其他列都是 0。因为我们的目标是只返回一行数据，最后一步就是要舍弃 DEPTNO 和 GROUP BY，只针对 CASE 表达式的结果执行 SUM 函数即可。

```
select sum(case when deptno=10 then 1 else 0 end) as deptno_10,
       sum(case when deptno=20 then 1 else 0 end) as deptno_20,
       sum(case when deptno=30 then 1 else 0 end) as deptno_30
  from emp

DEPTNO_10  DEPTNO_20  DEPTNO_30
--------- ---------- ----------
       3          5          6
```

对于这一类问题，还有一种可能的做法。

```
select max(case when deptno=10 then empcount else null end) as deptno_10,
       max(case when deptno=20 then empcount else null end) as deptno_20,
       max(case when deptno=30 then empcount else null end) as deptno_30
  from (
select deptno, count(*) as empcount
  from emp
 group by deptno
       ) x
```

上述方法用内嵌视图生成每个部门的员工总人数。主查询里的 CASE 表达式把行转换成列，将得到下面的结果。

```
DEPTNO_10  DEPTNO_20  DEPTNO_30
--------- ---------- ----------
        3       Null       Null
     Null          5       Null
     Null       Null          6
```

然后，调用 MAX 函数把几列合并为一行。

```
DEPTNO_10  DEPTNO_20  DEPTNO_30
--------- ---------- ----------
        3          5          6
```

12.2　变换结果集成多行

1. 问题

你想把行变换成列，并根据指定列的值来决定每一行原来的数据要被划分到新数据的哪一列。然而，与前一个实例不同的是，你需要输出的结果不止一行。

例如，你希望返回每个员工和他们的职位（对应 EMP 表的 JOB 列），你先查询得到了下面的结果集。

```
JOB       ENAME
--------- ----------
ANALYST   SCOTT
ANALYST   FORD
CLERK     SMITH
```

```
CLERK     ADAMS
CLERK     MILLER
CLERK     JAMES
MANAGER   JONES
MANAGER   CLARK
MANAGER   BLAKE
PRESIDENT KING
SALESMAN  ALLEN
SALESMAN  MARTIN
SALESMAN  TURNER
SALESMAN  WARD
```

你希望格式化上述结果集，为每一种职位生成一列新数据。

```
CLERKS ANALYSTS MGRS  PREZ SALES
------ -------- ----- ---- ------
MILLER FORD     CLARK KING TURNER
JAMES  SCOTT    BLAKE      MARTIN
ADAMS           JONES      WARD
SMITH                      ALLEN
```

2. 解决方案

与 12.1 节中的实例不同，本实例最后输出的结果集会多于一行。因此，12.1 节中实例的做法不适用于本例，因为如果针对每一个 JOB 列执行 MAX(ENAME) 的话，会导致每个 JOB 只对应一个 ENAME（也就是说，就会像 12.1 节中的实例那样只返回一行结果）。为了解决这一问题，我们必须保持每一个 JOB/ENAME 组合的唯一性。这样一来，再调用聚合函数剔除掉 Null 的话，就不会丢失任何一个 ENAME 了。

DB2、Oracle 和 SQL Server

使用窗口函数 ROW_NUMBER OVER 确保每一个 JOB/ENAME 组合都是唯一的。针对窗口函数的返回值执行 GROUP BY，并使用 CASE 表达式和聚合函数 MAX 实现结果集变换。

```
 1  select max(case when job='CLERK'
 2                  then ename else null end) as clerks,
 3         max(case when job='ANALYST'
 4                  then ename else null end) as analysts,
 5         max(case when job='MANAGER'
 6                  then ename else null end) as mgrs,
 7         max(case when job='PRESIDENT'
 8                  then ename else null end) as prez,
 9         max(case when job='SALESMAN'
10                  then ename else null end) as sales
11    from (
12  select job,
13         ename,
14         row_number()over(partition by job order by ename) rn
15    from emp
16         ) x
17   group by rn
```

PostgreSQL 和 MySQL

使用标量子查询基于 EMPNO 为每个员工排序。针对标量子查询的返回值执行 GROUP BY，并使

用 CASE 表达式和聚合函数 MAX 实现结果集变换。

```
 1  select max(case when job='CLERK'
 2                  then ename else null end) as clerks,
 3         max(case when job='ANALYST'
 4                  then ename else null end) as analysts,
 5         max(case when job='MANAGER'
 6                  then ename else null end) as mgrs,
 7         max(case when job='PRESIDENT'
 8                  then ename else null end) as prez,
 9         max(case when job='SALESMAN'
10                  then ename else null end) as sales
11    from (
12  select e.job,
13         e.ename,
14         (select count(*) from emp d
15           where e.job=d.job and e.empno < d.empno) as rnk
16    from emp e
17         ) x
18   group by rnk
```

3. 讨论

DB2、Oracle 和 SQL Server

首先，使用窗口函数 ROW_NUMBER OVER 确保每一个 JOB/ENAME 组合的唯一性。

```
select job,
       ename,
       row_number()over(partition by job order by ename) rn
  from emp

JOB       ENAME              RN
--------- ---------- ----------
ANALYST   FORD                1
ANALYST   SCOTT               2
CLERK     ADAMS               1
CLERK     JAMES               2
CLERK     MILLER              3
CLERK     SMITH               4
MANAGER   BLAKE               1
MANAGER   CLARK               2
MANAGER   JONES               3
PRESIDENT KING                1
SALESMAN  ALLEN               1
SALESMAN  MARTIN              2
SALESMAN  TURNER              3
SALESMAN  WARD                4
```

对于一种给定的职位，为其中的每一个 ENAME 安排一个唯一的"行编号"，这样即使出现了两个员工具有相同名字和职位的情况也不会有问题。这样做是为了既能基于行编号（RN）分组，又不会因为使用了 MAX 而遗漏掉任何一个员工。这是解决本问题最重要的一步。如果没有这一步，外层查询的聚合操作会剔除掉必要的行。试想一下，如果仍然采用 12.1 节中实例的做法，而不使用 ROW_NUMBER OVER 函数的话，会得到怎样的结果集。

```
select max(case when job='CLERK'
                then ename else null end) as clerks,
       max(case when job='ANALYST'
                then ename else null end) as analysts,
       max(case when job='MANAGER'
                then ename else null end) as mgrs,
       max(case when job='PRESIDENT'
                then ename else null end) as prez,
       max(case when job='SALESMAN'
                then ename else null end) as sales
  from emp

CLERKS     ANALYSTS   MGRS       PREZ       SALES
---------- ---------- ---------- ---------- ----------
SMITH      SCOTT      JONES      KING       WARD
```

很不幸，每一种 JOB 只会返回一行：ENAME 值最大的那个员工会被返回。做结果集变换的时候，使用 MIN 或 MAX 是为了从结果集里剔除掉 Null 值，而不是为了过滤掉一些 ENAME。继续阅读下面的讲解内容，相信你会更加清楚地理解这种状况是如何产生的。

下一步使用 CASE 表达式把 ENAME 分别放入各自的列（JOB）。

```
select rn,
       case when job='CLERK'
            then ename else null end as clerks,
       case when job='ANALYST'
            then ename else null end as analysts,
       case when job='MANAGER'
            then ename else null end as mgrs,
       case when job='PRESIDENT'
            then ename else null end as prez,
       case when job='SALESMAN'
            then ename else null end as sales
  from (
select job,
       ename,
       row_number()over(partition by job order by ename) rn
  from emp
       ) x

RN CLERKS     ANALYSTS   MGRS       PREZ       SALES
-- ---------- ---------- ---------- ---------- ----------
 1            FORD
 2            SCOTT
 1 ADAMS
 2 JAMES
 3 MILLER
 4 SMITH
 1                       BLAKE
 2                       CLARK
 3                       JONES
 1                                  KING
 1                                             ALLEN
 2                                             MARTIN
```

```
3                                            TURNER
4                                            WARD
```

现在，行已经变成了列，最后需要剔除掉 Null 值以提高结果集的可读性。为了剔除掉 Null，需要调用聚合函数 MAX，并基于 RN 执行 GROUP BY。（这里也可以使用 MIN 函数。选择 MAX 或者 MIN 没有区别，因为每个分组只包含一个值。）每个 RN/JOB/ENAME 组合只包含一个值。基于 RN 执行 GROUP BY，并且针对 CASE 表达式的返回值调用 MAX 函数，这能够确保每一次调用 MAX 函数都会从一个分组中取出一个 ENAME，而该分组内除了该 ENAME 之外，其他值都是 Null。

```
select max(case when job='CLERK'
                then ename else null end) as clerks,
       max(case when job='ANALYST'
                then ename else null end) as analysts,
       max(case when job='MANAGER'
                then ename else null end) as mgrs,
       max(case when job='PRESIDENT'
                then ename else null end) as prez,
       max(case when job='SALESMAN'
                then ename else null end) as sales
  from (
select job,
       ename,
       row_number()over(partition by job order by ename) rn
  from emp
       ) x
 group by rn

CLERKS ANALYSTS MGRS  PREZ SALES
------ -------- ----- ---- ------
MILLER FORD     CLARK KING TURNER
JAMES  SCOTT    BLAKE      MARTIN
ADAMS           JONES      WARD
SMITH                      ALLEN
```

使用 ROW_NUMBER OVER 生成唯一的组合对于格式化查询结果集非常有用。考虑如下所示的查询，它创建了一个分别以 DEPTNO 和 JOB 为维度展示员工的稀疏矩阵报表。

```
select deptno dno, job,
       max(case when deptno=10
                then ename else null end) as d10,
       max(case when deptno=20
                then ename else null end) as d20,
       max(case when deptno=30
                then ename else null end) as d30,
       max(case when job='CLERK'
                then ename else null end) as clerks,
       max(case when job='ANALYST'
                then ename else null end) as anals,
       max(case when job='MANAGER'
                then ename else null end) as mgrs,
       max(case when job='PRESIDENT'
                then ename else null end) as prez,
```

```
       max(case when job='SALESMAN'
                then ename else null end) as sales
  from (
select deptno,
       job,
       ename,
       row_number()over(partition by job order by ename) rn_job,
       row_number()over(partition by job order by ename) rn_deptno
  from emp
       ) x
 group by deptno, job, rn_deptno, rn_job
 order by 1

DNO JOB       D10    D20   D30    CLERKS ANALS MGRS  PREZ SALES
--- --------- ------ ----- ------ ------ ----- ----- ---- ------
 10 CLERK     MILLER              MILLER
 10 MANAGER   CLARK                            CLARK
 10 PRESIDENT KING                                   KING
 20 ANALYST          FORD                FORD
 20 ANALYST          SCOTT               SCOTT
 20 CLERK            ADAMS        ADAMS
 20 CLERK            SMITH        SMITH
 20 MANAGER          JONES                     JONES
 30 CLERK                  JAMES  JAMES
 30 MANAGER                BLAKE               BLAKE
 30 SALESMAN               ALLEN                          ALLEN
 30 SALESMAN               MARTIN                         MARTIN
 30 SALESMAN               TURNER                         TURNER
 30 SALESMAN               WARD                           WARD
```

通过改变分组列（即上述 SELECT 列表里的非聚合项），可以产生出不同格式的报表。这值得我们花时间去做一些实验，并深入理解 GROUP BY 子句包含的项目发生了改动之后，输出结果的格式将会发生什么变化。

PostgreSQL 和 MySQL

对于这两种数据库而言，做法和其他数据库基本相同，只是创建唯一 JOB/ENAME 组合的方法有所不同。首先使用标量子查询为每一个 JOB/ENAME 组合提供一个“行编号”或“排名”。

```
select e.job,
       e.ename,
       (select count(*) from emp d
         where e.job=d.job and e.empno < d.empno) as rnk
  from emp e

JOB       ENAME             RNK
--------- ---------- ----------
CLERK     SMITH               3
SALESMAN  ALLEN               3
SALESMAN  WARD                2
MANAGER   JONES               2
SALESMAN  MARTIN              1
MANAGER   BLAKE               1
```

```
MANAGER   CLARK          0
ANALYST   SCOTT          1
PRESIDENT KING           0
SALESMAN  TURNER         0
CLERK     ADAMS          2
CLERK     JAMES          1
ANALYST   FORD           0
CLERK     MILLER         0
```

为每一个 JOB/ENAME 组合安排一个唯一的“行编号”，这可以确保每一行都是唯一的。即使有两个员工名字相同，其职位也恰好一样，他们也不会共享同一个“行编号”。这是解决本问题最重要的一步。如果没有这一步，外层查询的聚合操作会剔除掉必要的行。试想一下，如果仍然采用 12.1 节中实例的做法，而不为每一个 JOB/ENAME 组合安排一个唯一的“行编号”的话，会得到怎样的结果集。

```
select max(case when job='CLERK'
                then ename else null end) as clerks,
       max(case when job='ANALYST'
                then ename else null end) as analysts,
       max(case when job='MANAGER'
                then ename else null end) as mgrs,
       max(case when job='PRESIDENT'
                then ename else null end) as prez,
       max(case when job='SALESMAN'
                then ename else null end) as sales
  from emp

CLERKS     ANALYSTS   MGRS       PREZ       SALES
---------- ---------- ---------- ---------- ----------
SMITH      SCOTT      JONES      KING       WARD
```

很不幸，每一种 JOB 只会返回一行：ENAME 值最大的那个员工会被返回。做结果集变换的时候，使用 MIN 或 MAX 是为了从结果集里剔除掉 Null 值，而不是为了过滤掉一些 ENAME。

现在，我们已经充分了解了“行编号”的作用，可以接着做下一步了。下一步使用 CASE 表达式把 ENAME 分别放入各自的列（JOB）。

```
select rnk,
       case when job='CLERK'
            then ename else null end as clerks,
       case when job='ANALYST'
            then ename else null end as analysts,
       case when job='MANAGER'
            then ename else null end as mgrs,
       case when job='PRESIDENT'
            then ename else null end as prez,
       case when job='SALESMAN'
            then ename else null end as sales
  from (
select e.job,
       e.ename,
       (select count(*) from emp d
         where e.job=d.job and e.empno < d.empno) as rnk
```

```
  from emp e
       ) x

RNK CLERKS ANALYSTS MGRS  PREZ SALES
--- ------ -------- ----- ---- ----------
  3 SMITH
  3                            ALLEN
  2                            WARD
  2                 JONES
  1                            MARTIN
  1                 BLAKE
  0                 CLARK
  1        SCOTT
  0                       KING
  0                            TURNER
  2 ADAMS
  1 JAMES
  0        FORD
  0 MILLER
```

现在，行已经变成了列，最后需要剔除掉 Null 值以提高结果集的可读性。为了剔除掉 Null，要调用聚合函数 MAX，并且基于 RNK 执行 GROUP BY。（这里也可以使用 MIN 函数，选择 MAX 或者 MIN 没有区别。）每个 RNK/JOB/ENAME 组合只存在一个值，因此调用了聚合函数之后，Null 值会被剔除掉。

```
select max(case when job='CLERK'
                then ename else null end) as clerks,
       max(case when job='ANALYST'
                then ename else null end) as analysts,
       max(case when job='MANAGER'
                then ename else null end) as mgrs,
       max(case when job='PRESIDENT'
                then ename else null end) as prez,
       max(case when job='SALESMAN'
                then ename else null end) as sales
  from (
select e.job,
       e.ename,
       (select count(*) from emp d
         where e.job=d.job and e.empno < d.empno) as rnk
  from emp e
       ) x
 group by rnk

CLERKS ANALYSTS MGRS  PREZ SALES
------ -------- ----- ---- ------
MILLER FORD     CLARK KING TURNER
JAMES  SCOTT    BLAKE      MARTIN
ADAMS           JONES      WARD
SMITH                      ALLEN
```

12.3 反向变换结果集

1. 问题

你想把列数据变换成行数据，考虑如下所示的结果集。

```
DEPTNO_10  DEPTNO_20  DEPTNO_30
---------- ---------- ----------
         3          5          6
```

你希望转换为：

```
DEPTNO COUNTS_BY_DEPT
------ --------------
    10              3
    20              5
    30              6
```

2. 解决方案

仔细查看上述的最终结果集，很容易想到我们只需针对 EMP 表执行 COUNT 和 GROUP BY 查询，就可以得到符合要求的结果集。然后，对于本实例而言，要先假设原始数据集没有被按照行形式存储，我们可以认为数据是存在于几个列中的。

为了把列数据变换成行数据，需要用到笛卡儿积。我们事先要知道有多少列需要被转换为行形式，因为创建笛卡儿积时用到的表表达式（table expression）必须有一个基数（cardinality），该基数至少要等于需要做变换的列的个数。

在这里，我们不必去创建一个“去规范化表”（denormalized table），只需重新使用 12.1 节中实例的代码生成一个“宽”结果集即可。完整的解决方案如下所示。

```
 1 select dept.deptno,
 2        case dept.deptno
 3             when 10 then emp_cnts.deptno_10
 4             when 20 then emp_cnts.deptno_20
 5             when 30 then emp_cnts.deptno_30
 6        end as counts_by_dept
 7   from (
 8 select sum(case when deptno=10 then 1 else 0 end) as deptno_10,
 9        sum(case when deptno=20 then 1 else 0 end) as deptno_20,
10        sum(case when deptno=30 then 1 else 0 end) as deptno_30
11   from emp
12        ) emp_cnts,
13        (select deptno from dept where deptno <= 30) dept
```

3. 讨论

内嵌视图 EMP_CNTS 就是上述的非规范化视图，即“宽”结果集，也就是变换前的列形式的数据，如下所示。

```
select sum(case when deptno=10 then 1 else 0 end) as deptno_10,
       sum(case when deptno=20 then 1 else 0 end) as deptno_20,
       sum(case when deptno=30 then 1 else 0 end) as deptno_30
  from emp
```

```
DEPTNO_10  DEPTNO_20  DEPTNO_30
--------- ---------- ----------
        3          5          6
```

因为上述数据分为 3 列存储，所以我们需要生成 3 行新数据。首先基于内嵌视图 EMP_CNTS 和一个至少有 3 行数据的表构造一个笛卡儿积。下面的代码借助 DEPT 表构造了一个笛卡儿积，DEPT 表里有 3 行数据。

```
select dept.deptno,
       emp_cnts.deptno_10,
       emp_cnts.deptno_20,
       emp_cnts.deptno_30
  from (
select sum(case when deptno=10 then 1 else 0 end) as deptno_10,
       sum(case when deptno=20 then 1 else 0 end) as deptno_20,
       sum(case when deptno=30 then 1 else 0 end) as deptno_30
  from emp
       ) emp_cnts,
       (select deptno from dept where deptno <= 30) dept

DEPTNO  DEPTNO_10  DEPTNO_20  DEPTNO_30
------ ---------- ---------- ----------
    10          3          5          6
    20          3          5          6
    30          3          5          6
```

笛卡儿积使得我们能够为内嵌视图 EMP_CNTS 的每一列返回一行数据。由于最终的结果集需要 DEPTNO 和该 DEPTNO 对应的员工人数，因此要用 CASE 表达式把每行 3 列变成每行 1 列。

```
select dept.deptno,
       case dept.deptno
            when 10 then emp_cnts.deptno_10
            when 20 then emp_cnts.deptno_20
            when 30 then emp_cnts.deptno_30
       end as counts_by_dept
  from (
select sum(case when deptno=10 then 1 else 0 end) as deptno_10,
       sum(case when deptno=20 then 1 else 0 end) as deptno_20,
       sum(case when deptno=30 then 1 else 0 end) as deptno_30
  from emp
       ) emp_cnts,
       (select deptno from dept where deptno <= 30) dept

DEPTNO COUNTS_BY_DEPT
------ --------------
    10              3
    20              5
    30              6
```

12.4 反向变换结果集成一列

1.问题

你想把一个查询结果合并成一列。例如，你希望返回 DEPTNO 等于 10 的全体员工的 ENAME、JOB 和 SAL，并且想把 3 列值合并成 1 列。你希望为每一个员工返回 3 行数据，员工之间以

空行分隔开。你希望得到如下所示的结果集。

```
EMPS
----------
CLARK
MANAGER
2450

KING
PRESIDENT
5000

MILLER
CLERK
1300
```

2.解决方案

关键在于使用笛卡儿积为每个员工返回 4 行数据。我们需要把每一列变成一行，并且在两个员工之间多留一个空白行。

DB2、Oracle 和 SQL Server

使用窗口函数 ROW_NUMBER OVER 基于 EMPNO 为每一行数据排名（1 ～ 4 行）。然后，使用 CASE 表达式把 3 列数据变成 1 列。

```
 1  select case rn
 2               when 1 then ename
 3               when 2 then job
 4               when 3 then cast(sal as char(4))
 5         end emps
 6    from (
 7  select e.ename,e.job,e.sal,
 8         row_number()over(partition by e.empno
 9                              order by e.empno) rn
10    from emp e,
11         (select *
12            from emp where job='CLERK') four_rows
13   where e.deptno=10
14         ) x
```

PostgreSQL 和 MySQL

本实例着重介绍如何使用窗口函数为每一行数据提供一个序号，该序号在之后的结果集变换操作中会被用到。在写作本书时，PostgreSQL 和 MySQL 尚未提供这样的窗口函数。

3. 讨论

DB2、Oracle 和 SQL Server

首先使用窗口函数 ROW_NUMBER OVER 为 DEPTNO 等于 10 每一位员工生成一个序号。

```
select e.ename,e.job,e.sal,
       row_number()over(partition by e.empno
                            order by e.empno) rn
  from emp e
 where e.deptno=10
```

```
ENAME      JOB              SAL         RN
---------- --------- ---------- ----------
CLARK      MANAGER         2450          1
KING       PRESIDENT       5000          1
MILLER     CLERK           1300          1
```

此时，上面的序号其实并没有意义。我们按照 EMPNO 分区，因而 DEPTNO 等于 10 的所有行的序号都是 1。如果引入笛卡儿积，序号的作用就显现出来了，如下所示。

```
select e.ename,e.job,e.sal,
       row_number()over(partition by e.empno
                            order by e.empno) rn
  from emp e,
       (select *
          from emp where job='CLERK') four_rows
 where e.deptno=10

ENAME      JOB              SAL         RN
---------- --------- ---------- ----------
CLARK      MANAGER         2450          1
CLARK      MANAGER         2450          2
CLARK      MANAGER         2450          3
CLARK      MANAGER         2450          4
KING       PRESIDENT       5000          1
KING       PRESIDENT       5000          2
KING       PRESIDENT       5000          3
KING       PRESIDENT       5000          4
MILLER     CLERK           1300          1
MILLER     CLERK           1300          2
MILLER     CLERK           1300          3
MILLER     CLERK           1300          4
```

现在我们需要暂停一下，仔细推敲两个关键的要点。

- 每个员工的 RN 值不再是 1，现在它变成了从 1 到 4 循环出现的序列值，原因在于窗口函数会在 FROM 和 WHERE 子句之后才被评估执行。因此，按照 EMPNO 分区就导致当遇到一个新员工时，RN 值会被重置为 1。
- 内嵌视图 FOUR_ROWS 的存在只是为了返回一个包含 4 行数据的结果集。它的作用仅限于此。我们希望为每一列（ENAME、JOB 和 SAL）返回一行，然后再加上一个空行。

现在，最困难的工作已经完成了，剩下的就是使用 CASE 表达式把每个员工的 ENAME、JOB 和 SAL 并入一列（为了保证 CASE 成功执行，我们还需要把 SAL 转换成字符串）。

```
select case rn
            when 1 then ename
            when 2 then job
            when 3 then cast(sal as char(4))
       end emps
  from (
select e.ename,e.job,e.sal,
       row_number()over(partition by e.empno
                           order by e.empno) rn
  from emp e,
       (select *
```

```
          from emp where job='CLERK') four_rows
 where e.deptno=10
       ) x

   EMPS
   ----------
   CLARK
   MANAGER
   2450

   KING
   PRESIDENT
   5000

   MILLER
   CLERK
   1300
```

12.5　删除重复数据

1. 问题

你正在生成一个报表，当相邻两行的某列出现了相同值时，你希望那个值只显示一次。例如，你想从 EMP 表中提取出 DEPTNO 和 ENAME，希望按照 DEPTNO 对所有的行进行分组，并且希望每个 DEPTNO 只显示一次。你希望返回如下所示的结果集。

```
DEPTNO ENAME
------ ---------
    10 CLARK
       KING
       MILLER
    20 SMITH
       ADAMS
       FORD
       SCOTT
       JONES
    30 ALLEN
       BLAKE
       MARTIN
       JAMES
       TURNER
       WARD
```

2. 解决方案

这是一个很简单的格式化问题，Oracle 提供的窗口函数 LAG OVER 能很容易地解决这一问题。当然，使用标量子查询和其他窗口函数也能达到同样目的（对于非 Oracle 用户只能如此），但这里最方便的做法是使用 LAG OVER 函数。

DB2 和 SQL Server

使用窗口函数 MIN OVER 为每个 DEPTNO 找出最小的 EMPNO，然后使用 CASE 表达式来“涂改”那些 EMPNO 不等于该最小值的行。

```
 1  select case when empno=min_empno
 2              then deptno else null
 3         end deptno,
 4         ename
 5    from (
 6  select deptno,
 7         min(empno)over(partition by deptno) min_empno,
 8         empno,
 9         ename
10    from emp
11         ) x
```

Oracle

使用窗口函数 LAG OVER 访问当前行的前一行，为每个分区间找出第一个 DEPTNO。

```
1 select to_number(
2           decode(lag(deptno)over(order by deptno),
3                 deptno,null,deptno)
4        ) deptno, ename
5   from emp
```

PostgreSQL 和 MySQL

本实例着重介绍如何使用窗口函数方便地访问到当前行前面和后面的行数据。在写作本书时，这些数据库尚未支持这类窗口函数。

3. 讨论

DB2 和 SQL Server

首先，使用窗口函数 MIN OVER 找出每个 DEPTNO 对应的最小 EMPNO 值。

```
select deptno,
       min(empno)over(partition by deptno) min_empno,
       empno,
       ename
  from emp

DEPTNO MIN_EMPNO       EMPNO ENAME
------ ---------- ---------- ----------
    10       7782       7782 CLARK
    10       7782       7839 KING
    10       7782       7934 MILLER
    20       7369       7369 SMITH
    20       7369       7876 ADAMS
    20       7369       7902 FORD
    20       7369       7788 SCOTT
    20       7369       7566 JONES
    30       7499       7499 ALLEN
    30       7499       7698 BLAKE
    30       7499       7654 MARTIN
    30       7499       7900 JAMES
    30       7499       7844 TURNER
    30       7499       7521 WARD
```

下一步也是最后一步，使用 CASE 表达式删除重复的 DEPTNO。如果一个员工的 EMPNO 和

MIN_EMPNO 相等，则返回 DEPTNO，否则返回 Null。

```
select case when empno=min_empno
            then deptno else null
       end deptno,
       ename
  from (
select deptno,
       min(empno)over(partition by deptno) min_empno,
       empno,
       ename
  from emp
       ) x

DEPTNO ENAME
------ ----------
    10 CLARK
       KING
       MILLER
    20 SMITH
       ADAMS
       FORD
       SCOTT
       JONES
    30 ALLEN
       BLAKE
       MARTIN
       JAMES
       TURNER
       WARD
```

Oracle

首先，使用窗口函数 LAG OVER 为每一行返回前一行的 DEPTNO。

```
select lag(deptno)over(order by deptno) lag_deptno,
       deptno,
       ename
  from emp

LAG_DEPTNO     DEPTNO ENAME
---------- ---------- ----------
                   10 CLARK
        10         10 KING
        10         10 MILLER
        10         20 SMITH
        20         20 ADAMS
        20         20 FORD
        20         20 SCOTT
        20         20 JONES
        20         30 ALLEN
        30         30 BLAKE
        30         30 MARTIN
        30         30 JAMES
        30         30 TURNER
        30         30 WARD
```

如果仔细观察以上结果集的话，很容易区分出哪些行的 DEPTNO 和 LAG_DEPTNO 相等。对于这些行，我们希望把 DEPTNO 显示为 Null。我们可以借助 DECODE 函数做到这一点（TO_NUMBER 函数可以把 DEPTNO 转换为数字）。

```
select to_number(
           decode(lag(deptno)over(order by deptno),
                  deptno,null,deptno)
       ) deptno, ename
  from emp

DEPTNO ENAME
------ ----------
    10 CLARK
       KING
       MILLER
    20 SMITH
       ADAMS
       FORD
       SCOTT
       JONES
    30 ALLEN
       BLAKE
       MARTIN
       JAMES
       TURNER
       WARD
```

12.6 变换结果集以实现跨行计算

1. 问题

你希望计算来自多行的数据。为了让这个任务变得容易些，你希望将那些行都转换为列，这样你需要的所有值都会出现在同一行里。

在本书的示例数据中，DEPTNO 20 是工资总额最高的部门，我们可以通过下列查询来确认一下数据。

```
select deptno, sum(sal) as sal
  from emp
 group by deptno

DEPTNO        SAL
------ ----------
    10       8750
    20      10875
    30       9400
```

你希望计算出上述 DEPTNO 20 和 DEPTNO 10 之间的工资总额的差值，以及上述 DEPTNO 20 和 DEPTNO 30 之间的工资总额差值。

2. 解决问题

使用聚合函数 SUM 和 CASE 表达式计算工资总额，然后在 SELECT 列表里做差值计算。

```
1  select d20_sal - d10_sal as d20_10_diff,
2         d20_sal - d30_sal as d20_30_diff
3    from (
4  select sum(case when deptno=10 then sal end) as d10_sal,
5         sum(case when deptno=20 then sal end) as d20_sal,
6         sum(case when deptno=30 then sal end) as d30_sal
7    from emp
8         ) totals_by_dept
```

3. 讨论

首先，使用 CASE 表达式把每个 DEPTNO 对应的工资从行形式变换为列形式。

```
select case when deptno=10 then sal end as d10_sal,
       case when deptno=20 then sal end as d20_sal,
       case when deptno=30 then sal end as d30_sal
  from emp

D10_SAL    D20_SAL    D30_SAL
------- ---------- ----------
               800
                         1600
                         1250
              2975
                         1250
                         2850
   2450
              3000
   5000
                         1500
              1100
                          950
              3000
   1300
```

然后，在每一个 CASE 表达式里调用聚合函数 SUM 计算出每个 DEPTNO 对应的工资总额。

```
select sum(case when deptno=10 then sal end) as d10_sal,
       sum(case when deptno=20 then sal end) as d20_sal,
       sum(case when deptno=30 then sal end) as d30_sal
  from emp

D10_SAL    D20_SAL    D30_SAL
------- ---------- ----------
   8750      10875       9400
```

最后，只要把上述 SQL 查询放入到内嵌视图里，并在外层查询中执行减法运算即可。

12.7 创建固定大小的数据桶

1. 问题

你想把数据放入若干个大小固定的桶（bucket）里，每个桶的元素个数是事先定好的。桶的个数可能是不确定的，但你希望确保每个桶有 5 个元素。例如，你希望基于 EMPNO 值为 EMP 表里的员工分组，一组最多 5 人，结果集显示如下。

```
GRP      EMPNO ENAME
--- ---------- -------
  1       7369 SMITH
  1       7499 ALLEN
  1       7521 WARD
  1       7566 JONES
  1       7654 MARTIN
  2       7698 BLAKE
  2       7782 CLARK
  2       7788 SCOTT
  2       7839 KING
  2       7844 TURNER
  3       7876 ADAMS
  3       7900 JAMES
  3       7902 FORD
  3       7934 MILLER
```

2. 解决方案

对于提供了排名函数的数据库而言，很容易解决本问题。在为每一行数据生成了一个序号之后，创建含有 5 个元素的桶的问题就变成了简单的除法问题。做过除法之后，我们只须针对商值向上取整即可。

DB2、Oracle 和 SQL Server

使用窗口函数 ROW_NUMBER OVER，基于 EMPNO 为每个员工生成一个序号。然后用该序号除以 5 即可实现分组（SQL Server 需要调用 CEILING 函数，而不是 CEIL 函数）。

```
1 select ceil(row_number()over(order by empno)/5.0) grp,
2        empno,
3        ename
4   from emp
```

PostgreSQL 和 MySQL

使用标量子查询为每个 EMPNO 生成一个序号，然后用该序号除以 5 以创建分组。

```
1  select ceil(rnk/5.0) as grp,
2         empno, ename
3    from (
4  select e.empno, e.ename,
5         (select count(*) from emp d
6           where e.empno < d.empno)+1 as rnk
7    from emp e
8         ) x
9   order by grp
```

3.讨论

DB2、Oracle 和 SQL Server

按照 EMPNO 排序后，窗口函数 ROW_NUMBER OVER 为每一行分配了一个排名或“行号”。

```
select row_number()over(order by empno) rn,
       empno,
       ename
  from emp
```

```
RN      EMPNO ENAME
-- ---------- ----------
 1       7369 SMITH
 2       7499 ALLEN
 3       7521 WARD
 4       7566 JONES
 5       7654 MARTIN
 6       7698 BLAKE
 7       7782 CLARK
 8       7788 SCOTT
 9       7839 KING
10       7844 TURNER
11       7876 ADAMS
12       7900 JAMES
13       7902 FORD
14       7934 MILLER
```

将 ROW_NUMBER OVER 函数的返回值除以 5 之后，下一步要调用函数 CEIL（或 CEILING）。理论上，除以 5，会把每 5 行数据划入一组。也就是说，会有 5 个值小于或等于 1，同时会有 5 个值大于 1 但小于或等于 2，剩下的一组则大于 2 但小于或等于 3（这一组由最后的 4 行数据构成，因为 EMP 表一共包含 14 行数据，并非 5 的整数倍）。

CEIL 函数将返回大于参数值的最小的整数。我们需要这样做，因为每一组的编号都是一个整数值。除法计算的结果以及 CEIL 函数的返回值显示如下。你可以按照从左到右、从 RN 到 DIVISION 再到 GRP 的顺序试着做一下运算。

```
select row_number()over(order by empno) rn,
       row_number()over(order by empno)/5.0 division,
       ceil(row_number()over(order by empno)/5.0) grp,
       empno,
       ename
  from emp

RN   DIVISION GRP EMPNO ENAME
-- ---------- --- ----- ----------
 1         .2   1  7369 SMITH
 2         .4   1  7499 ALLEN
 3         .6   1  7521 WARD
 4         .8   1  7566 JONES
 5          1   1  7654 MARTIN
 6        1.2   2  7698 BLAKE
 7        1.4   2  7782 CLARK
 8        1.6   2  7788 SCOTT
 9        1.8   2  7839 KING
10          2   2  7844 TURNER
11        2.2   3  7876 ADAMS
12        2.4   3  7900 JAMES
13        2.6   3  7902 FORD
14        2.8   3  7934 MILLER
```

PostgreSQL 和 MySQL

首先借助标量子查询基于 EMPNO 为每一行生成一个序号。

```
select (select count(*) from emp d
         where e.empno < d.empno)+1 as rnk,
       e.empno, e.ename
  from emp e
 order by 1

RNK      EMPNO ENAME
--- ---------- ----------
  1       7934 MILLER
  2       7902 FORD
  3       7900 JAMES
  4       7876 ADAMS
  5       7844 TURNER
  6       7839 KING
  7       7788 SCOTT
  8       7782 CLARK
  9       7698 BLAKE
 10       7654 MARTIN
 11       7566 JONES
 12       7521 WARD
 13       7499 ALLEN
 14       7369 SMITH
```

上述 RNK 值除以 5 之后，下一步要调用函数 CEIL。理论上，除以 5 会把每 5 行数据划入一组。也就是说，会有 5 个值小于或等于 1，同时会有 5 个值大于 1 但小于或等于 2，剩下的一组则大于 2 但小于或等于 3（这一组由最后的 4 行数据构成，因为 EMP 表一共包含 14 行数据，并非 5 的整数倍）。除法计算的结果以及 CEIL 函数的返回值显示如下。你可以按照从左到右、从 RNK 到 DIVISION 再到 GRP 的顺序试着做一下运算。

```
select rnk,
       rnk/5.0 as division,
       ceil(rnk/5.0) as grp,
       empno, ename
  from (
select e.empno, e.ename,
       (select count(*) from emp d
         where e.empno < d.empno)+1 as rnk
  from emp e
       ) x
 order by 1

RNK   DIVISION GRP EMPNO ENAME
--- ---------- --- ----- -------
  1         .2   1  7934 MILLER
  2         .4   1  7902 FORD
  3         .6   1  7900 JAMES
  4         .8   1  7876 ADAMS
  5          1   1  7844 TURNER
  6        1.2   2  7839 KING
  7        1.4   2  7788 SCOTT
  8        1.6   2  7782 CLARK
  9        1.8   2  7698 BLAKE
 10          2   2  7654 MARTIN
 11        2.2   3  7566 JONES
```

```
12        2.4   3  7521 WARD
13        2.6   3  7499 ALLEN
14        2.8   3  7369 SMITH
```

12.8 创建预定数目的桶

1. 问题

你想把你的数据分别放入到数目固定的桶里面去。例如，你希望把 EMP 表里的员工分别放入到 4 个桶里，结果集应该如下所示。

```
GRP EMPNO ENAME
--- ----- ---------
  1  7369 SMITH
  1  7499 ALLEN
  1  7521 WARD
  1  7566 JONES
  2  7654 MARTIN
  2  7698 BLAKE
  2  7782 CLARK
  2  7788 SCOTT
  3  7839 KING
  3  7844 TURNER
  3  7876 ADAMS
  4  7900 JAMES
  4  7902 FORD
  4  7934 MILLER
```

本实例和 12.7 节中的实例恰好相反。在 12.7 节中的实例里，桶的个数并没有限制，但每个桶的元素个数却是事先定好的。对于本实例而言，你不在乎每个桶里有多少个元素，但需要创建固定数目（数目已知）的桶。

2. 解决方案

对于那些提供了专有函数帮助我们创建“桶”的数据库而言，很容易解决本问题。但是，如果数据库不提供这类函数，则只好为每一行生成一个序号，然后针对该序号和 n 执行模运算以决定把某一行放入哪个桶，此处的 n 代表我们希望创建的桶的个数。针对部分提供了这些函数的数据库，本解决方案将借助窗口函数 NTILE 创建数目固定的桶。NTILE 负责把排好序的集合分别放入到指定数目的桶里去，每一个元素都必然会被分配到某个桶中。这恰好与前面给出的我们所期望的结果集相一致，桶 1 和桶 2 分别有 4 行数据，而桶 3 和桶 4 却只有 3 行。如果数据库不支持 NTILE，也不必担心。本实例的主要目的是创建固定数目的桶，我们没必要执着于一定要把某一行放进哪个桶。

DB2

使用窗口函数 ROW_NUMBER OVER 基于 EMPNO 为每一行生成一个序号，然后针对该序号和 4 执行模运算以创建 4 个桶。

```
1 select mod(row_number()over(order by empno),4)+1 grp,
2        empno,
3        ename
4   from emp
5  order by 1
```

Oracle 和 SQL Server

DB2 解决方案也适用于这两种数据库。除此之外，我们还可以使用窗口函数 NTILE 创建 4 个桶（这种做法更简单）。

```
1  select ntile(4)over(order by empno) grp,
2         empno,
3         ename
4    from emp
```

MySQL 和 PostgreSQL

使用自连接基于 EMPNO 为每一行生成一个序号，然后针对该序号和 4 执行模运算以创建桶。

```
1  select mod(count(*),4)+1 as grp,
2          e.empno,
3          e.ename
4    from emp e, emp d
5   where e.empno >= d.empno
6   group by e.empno,e.ename
7   order by 1
```

3. 讨论

DB2

首先使用窗口函数 ROW_NUMBER OVER 基于 EMPNO 为每一行生成一个序号。

```
select row_number()over(order by empno) grp,
       empno,
       ename
  from emp

GRP EMPNO ENAME
--- ----- ------
  1  7369 SMITH
  2  7499 ALLEN
  3  7521 WARD
  4  7566 JONES
  5  7654 MARTIN
  6  7698 BLAKE
  7  7782 CLARK
  8  7788 SCOTT
  9  7839 KING
 10  7844 TURNER
 11  7876 ADAMS
 12  7900 JAMES
 13  7902 FORD
 14  7934 MILLER
```

现在每一行都分配好了序号，下面要调用模运算函数 MOD 创建 4 个桶。

```
select mod(row_number()over(order by empno),4) grp,
       empno,
       ename
  from emp
```

```
GRP EMPNO ENAME
--- ----- ------
  1  7369 SMITH
  2  7499 ALLEN
  3  7521 WARD
  0  7566 JONES
  1  7654 MARTIN
  2  7698 BLAKE
  3  7782 CLARK
  0  7788 SCOTT
  1  7839 KING
  2  7844 TURNER
  3  7876 ADAMS
  0  7900 JAMES
  1  7902 FORD
  2  7934 MILLER
```

最后为 GRP 加 1，这样桶的编号才会从 1 而不是 0 开始，然后还要使用 ORDER BY 基于 GRP 排序。

Oracle 和 SQL Server

NTILE 函数独自完成了全部工作。我们只需要传递一个参数告诉 NTILE 我们想要几个桶，然后等着看结果即可。

MySQL 和 PostgreSQL

首先和 EMP 表生成笛卡儿积，这样每一个 EMPNO 就能够和其他的任意 EMPNO 进行比较了，下面只截取了该笛卡儿积的一部分，因为全部的返回值会有 196 行（14 乘以 14）。

```
select e.empno,
       e.ename,
       d.empno,
       d.ename
  from emp e, emp d

EMPNO ENAME           EMPNO ENAME
----- ---------- ---------- ---------
 7369 SMITH            7369 SMITH
 7369 SMITH            7499 ALLEN
 7369 SMITH            7521 WARD
 7369 SMITH            7566 JONES
 7369 SMITH            7654 MARTIN
 7369 SMITH            7698 BLAKE
 7369 SMITH            7782 CLARK
 7369 SMITH            7788 SCOTT
 7369 SMITH            7839 KING
 7369 SMITH            7844 TURNER
 7369 SMITH            7876 ADAMS
 7369 SMITH            7900 JAMES
 7369 SMITH            7902 FORD
 7369 SMITH            7934 MILLER
 …
```

从上述结果集中可以看到，SMITH 的 EMPNO 会和 EMP 表中每一个 EMPNO 进行比较（每个员工的 EMPNO 都可以和所有其他员工的 EMPNO 进行比较）。下一步要限定笛卡儿积的结果，只有那些 EMPNO 大于或等于其他 EMPNO 的行才会被保留下来。部分结果集如下所示（因为总共有 105 行）。

```
select e.empno,
       e.ename,
       d.empno,
       d.ename
  from emp e, emp d
 where e.empno >= d.empno

EMPNO ENAME           EMPNO ENAME
----- ---------- ---------- ----------
 7934 MILLER           7934 MILLER
 7934 MILLER           7902 FORD
 7934 MILLER           7900 JAMES
 7934 MILLER           7876 ADAMS
 7934 MILLER           7844 TURNER
 7934 MILLER           7839 KING
 7934 MILLER           7788 SCOTT
 7934 MILLER           7782 CLARK
 7934 MILLER           7698 BLAKE
 7934 MILLER           7654 MARTIN
 7934 MILLER           7566 JONES
 7934 MILLER           7521 WARD
 7934 MILLER           7499 ALLEN
 7934 MILLER           7369 SMITH
 ...
 7499 ALLEN            7499 ALLEN
 7499 ALLEN            7369 SMITH
 7369 SMITH            7369 SMITH
```

以上输出结果并非全部的结果集，它只包括来自 EMP E 表的 MILLER、ALLEN 和 SMITH。我只是想告诉你 WHERE 子句会怎样限制笛卡儿积的结果。因为 WHERE 子句的 EMPNO 过滤条件是“大于或等于”，这意味着每个员工至少有一行查询结果，因为每个 EMPNO 都等于它自身。但是，为什么 SMITH 只有 1 行（在结果集的左边），ALLEN 有 2 行，而 MILLER 则有 14 行呢？原因就在于 WHERE 子句里那个关于 EMPNO 的复合条件：“大于或等于”。对于 SMITH 而言，没有比 7369 小的 EMPNO，因此只有一行数据返回。对于 ALLEN 而言，很显然他的 EMPNO 等于其自身（因此返回了该行数据），但 7499 也大于 7369（SMITH 的 EMPNO），因此会返回两行数据。对于 MILLER 而言，他的 EMPNO 7934 大于 EMP 表中所有其他 EMPNO（并且显然也等于其自身），因此会返回 14 行数据。

现在我们可以比较每一个 EMPNO，并且决定哪些大于其他的值。在自连接查询里使用聚合函数 COUNT 返回一个更具表现力的结果集。

```
select count(*) as grp,
       e.empno,
       e.ename
  from emp e, emp d
 where e.empno >= d.empno
 group by e.empno,e.ename
 order by 1

GRP      EMPNO ENAME
--- ---------- ----------
  1       7369 SMITH
  2       7499 ALLEN
  3       7521 WARD
```

```
  4       7566 JONES
  5       7654 MARTIN
  6       7698 BLAKE
  7       7782 CLARK
  8       7788 SCOTT
  9       7839 KING
 10       7844 TURNER
 11       7876 ADAMS
 12       7900 JAMES
 13       7902 FORD
 14       7934 MILLER
```

现在每一行都有了一个序号，先对 GRP 和 4 执行模运算，然后再加上 1，这样就创建出了 4 个桶（加 1 是为了让桶从 1 开始，而不是 0）。针对 GRP 使用 ORDER BY 子句进行排序，这样就能以合适的顺序输出最终结果。

```
select mod(count(*),4)+1 as grp,
       e.empno,
       e.ename
  from emp e, emp d
 where e.empno >= d.empno
 group by e.empno,e.ename
 order by 1

GRP      EMPNO ENAME
--- ---------- ---------
  1       7900 JAMES
  1       7566 JONES
  1       7788 SCOTT
  2       7369 SMITH
  2       7902 FORD
  2       7654 MARTIN
  2       7839 KING
  3       7499 ALLEN
  3       7698 BLAKE
  3       7934 MILLER
  3       7844 TURNER
  4       7521 WARD
  4       7782 CLARK
  4       7876 ADAMS
```

12.9 创建水平直方图

1. 问题

你想用 SQL 创建水平直方图。例如，你希望以水平直方图的形式显示每个部门的员工人数，用一个“*”代表一个员工。你希望返回如下所示的结果集。

```
DEPTNO CNT
------ ----------
    10 ***
    20 *****
    30 ******
```

2. 解决方案

解决本问题的关键是使用聚合函数 COUNT 和 GROUP BY DEPTNO 计算每个 DEPTNO 对应的员工人数。然后，把 COUNT 的返回值传递给字符串函数以生成一系列的“*”字符。

DB2

使用 REPEAT 函数生成直方图。

```
1 select deptno,
2        repeat('*',count(*)) cnt
3   from emp
4  group by deptno
```

Oracle、PostgreSQL 和 MySQL

使用 LPAD 函数生成所需的字符串“*”。

```
1 select deptno,
2        lpad('*',count(*),'*') as cnt
3   from emp
4  group by deptno
```

SQL Server

使用 REPLICATE 函数生成直方图。

```
1 select deptno,
2        replicate('*',count(*)) cnt
3   from emp
4  group by deptno
```

3. 讨论

对于所有的数据库而言，这里用到的方法很类似。唯一的不同之处在于用来生成代表员工人数的 * 字符串的函数。下面的讨论以 Oracle 解决方案为例，但也包含了其他的解决方案。

首先计算出每个部门的员工人数。

```
select deptno,
       count(*)
  from emp
 group by deptno

DEPTNO   COUNT(*)
------ ----------
    10          3
    20          5
    30          6
```

然后，用 COUNT(*) 的返回值控制每个部门对应的 * 字符的个数。只要把 COUNT(*) 作为参数传递给 LPAD 函数就可以生成所需数目的 *。

```
select deptno,
       lpad('*',count(*),'*') as cnt
  from emp
 group by deptno
```

```
DEPTNO CNT
------ ----------
    10 ***
    20 *****
    30 ******
```

PostgreSQL 用户需要明确地把 COUNT(*) 的返回值先转换成整数，如下所示。

```
select deptno,
       lpad('*',count(*)::integer,'*') as cnt
  from emp
 group by deptno

DEPTNO CNT
------ ----------
    10 ***
    20 *****
    30 ******
```

上面的 CAST 函数调用是必须的，因为 PostgreSQL 要求 LPAD 的参数为整数。

12.10 创建垂直直方图

1. 问题

你想生成一个从下向上增长的直方图。例如，你希望以垂直直方图的方式显示每个部门的员工人数，每个 * 代表一个员工。你希望返回如下所示的结果集。

```
D10 D20 D30
--- --- ---
        *
    *   *
    *   *
*   *   *
*   *   *
*   *   *
```

2. 解决方案

我们将以 12.2 节中的方法为基础解决本问题。

DB2、Oracle 和 SQL Server

使用窗口函数 ROW_NUMBER OVER 为每个 DEPTNO 的每一个 * 生成唯一的序号。使用聚合函数 MAX 变换结果集，并针对 ROW_NUMBER OVER 函数的返回值执行 GROUP BY。SQL Server 用户不要在该 ORDER BY 子句中使用 DESC。

```
1 select max(deptno_10) d10,
2        max(deptno_20) d20,
3        max(deptno_30) d30
4   from (
5 select row_number()over(partition by deptno order by empno) rn,
6        case when deptno=10 then '*' else null end deptno_10,
7        case when deptno=20 then '*' else null end deptno_20,
8        case when deptno=30 then '*' else null end deptno_30
9   from emp
```

```
10         ) x
11   group by rn
12   order by 1 desc, 2 desc, 3 desc
```

PostgreSQL 和 MySQL

使用标量子查询为每个 DEPTNO 的每一个 * 生成唯一的序号。针对内嵌视图 X 的返回值调用聚合函数 MAX，同时也针对 RNK 执行 GROUP BY 以实现结果集变换。MySQL 用户不要在 ORDER BY 子句中使用 DESC。

```
 1  select max(deptno_10) as d10,
 2         max(deptno_20) as d20,
 3         max(deptno_30) as d30
 4    from (
 5  select case when e.deptno=10 then '*' else null end deptno_10,
 6         case when e.deptno=20 then '*' else null end deptno_20,
 7         case when e.deptno=30 then '*' else null end deptno_30,
 8         (select count(*) from emp d
 9           where e.deptno=d.deptno and e.empno < d.empno ) as rnk
10    from emp e
11         ) x
12   group by rnk
13   order by 1 desc, 2 desc, 3 desc
```

3. 讨论

DB2、Oracle 和 SQL Server

首先使用窗口函数 ROW_NUMBER 为每个 DEPTNO 的每一个 * 生成唯一的序号。使用 CASE 表达式为每个部门的每个员工返回一个 *。

```
select row_number()over(partition by deptno order by empno) rn,
       case when deptno=10 then '*' else null end deptno_10,
       case when deptno=20 then '*' else null end deptno_20,
       case when deptno=30 then '*' else null end deptno_30
  from emp

RN DEPTNO_10  DEPTNO_20  DEPTNO_30
-- ---------- ---------- ---------
 1 *
 2 *
 3 *
 1          *
 2          *
 3          *
 4          *
 5          *
 1                       *
 2                       *
 3                       *
 4                       *
 5                       *
 6                       *
```

下一步也是最后一步，针对每个 CASE 表达式调用聚合函数 MAX，并基于 RN 分组剔除掉 Null 值。以 ASC 或 DESC 方式排序取决于数据库如何对 Null 值排序。

```
select max(deptno_10) d10,
       max(deptno_20) d20,
       max(deptno_30) d30
  from (
select row_number()over(partition by deptno order by empno) rn,
       case when deptno=10 then '*' else null end deptno_10,
       case when deptno=20 then '*' else null end deptno_20,
       case when deptno=30 then '*' else null end deptno_30
  from emp
       ) x
 group by rn
 order by 1 desc, 2 desc, 3 desc

D10 D20 D30
--- --- ---
        *
    *   *
    *   *
*   *   *
*   *   *
*   *   *
```

PostgreSQL 和 MySQL

首先，使用标量子查询为每个 DEPTNO 的每一个 * 生成唯一的序号。该标量子查询基于 EMPNO 为每个 DEPTNO 的员工生成序号，因此不可能有重复项。使用 CASE 表达式为每个部门的每一个员工返回一个 *。

```
select case when e.deptno=10 then '*' else null end deptno_10,
       case when e.deptno=20 then '*' else null end deptno_20,
       case when e.deptno=30 then '*' else null end deptno_30,
       (select count(*) from emp d
         where e.deptno=d.deptno and e.empno < d.empno ) as rnk
  from emp e

DEPTNO_10  DEPTNO_20  DEPTNO_30         RNK
---------- ---------- ---------- ----------
           *                              4
                      *                   5
                      *                   4
           *                              3
                      *                   3
                      *                   2
*                                         2
           *                              2
*                                         1
                      *                   1
           *                              1
                      *                   0
           *                              0
*                                         0
```

然后，针对每个 CASE 表达式调用聚合函数 MAX。这样一来，按照 RNK 分组后就能够从结果集中剔除掉 Null 值了。以 ASC 或 DESC 方式排序取决于数据库如何对 Null 值排序。

```
select max(deptno_10) as d10,
       max(deptno_20) as d20,
       max(deptno_30) as d30
  from (
select case when e.deptno=10 then '*' else null end deptno_10,
       case when e.deptno=20 then '*' else null end deptno_20,
       case when e.deptno=30 then '*' else null end deptno_30,
       (select count(*) from emp d
         where e.deptno=d.deptno and e.empno < d.empno ) as rnk
  from emp e
       ) x
 group by rnk
 order by 1 desc, 2 desc, 3 desc

D10 D20 D30
--- --- ---
        *
    *   *
    *   *
*   *   *
*   *   *
*   *   *
```

12.11 返回非分组列

1. 问题

你正在执行 GROUP BY 查询，并希望通过 SELECT 列表返回一些列，但这些列却不会出现在 GROUP BY 子句里。这通常无法办到，因为不能保证这些列在每个分组里都有唯一的值。

假设你希望找出每个部门工资最高和最低的员工，同时也希望找出每个职位对应的工资最高和最低的员工。你想查看每个员工的名字、部门、职位以及工资。你希望返回如下所示的结果集。

```
DEPTNO ENAME  JOB         SAL DEPT_STATUS     JOB_STATUS
------ ------ --------- ----- --------------- --------------
    10 MILLER CLERK      1300 LOW SAL IN DEPT TOP SAL IN JOB
    10 CLARK  MANAGER    2450                 LOW SAL IN JOB
    10 KING   PRESIDENT  5000 TOP SAL IN DEPT TOP SAL IN JOB
    20 SCOTT  ANALYST    3000 TOP SAL IN DEPT TOP SAL IN JOB
    20 FORD   ANALYST    3000 TOP SAL IN DEPT TOP SAL IN JOB
    20 SMITH  CLERK       800 LOW SAL IN DEPT LOW SAL IN JOB
    20 JONES  MANAGER    2975                 TOP SAL IN JOB
    30 JAMES  CLERK       950 LOW SAL IN DEPT
    30 SALESMAN          1250                 LOW SAL IN JOB
    30 WARD   SALESMAN   1250                 LOW SAL IN JOB
    30 ALLEN  SALESMAN   1600                 TOP SAL IN JOB
    30 BLAKE  MANAGER    2850 TOP SAL IN DEPT
```

不幸的是，如果把上述所有列都放入 SELECT 子句的话，将会破坏分组操作。考虑如下的例子。员工 KING 的工资最高，你想用下列查询验证这一点。

```
select ename,max(sal)
  from emp
 group by ename
```

以上查询将返回 EMP 表的全部 14 行数据，而不只是 KING 及其工资。之所以会这样，正是因为那个分组操作的存在：它会针对每个 ENAME 调用 MAX(SAL) 函数。因此，上述 SQL 语句看起来好像能够“找出工资最高的员工”，但实际的执行结果却是“找出了 EMP 表里每个 ENAME 对应的最高工资”。因此，本实例将介绍一种能够查询到 ENAME 却不必把 ENAME 放入 GROUP BY 子句的方法。

2. 解决方案

使用内嵌视图找出每个 DEPTNO 和 JOB 对应的最高和最低的工资。然后，筛选出工资等于这些值的员工。

DB2、Oracle 和 SQL Server

使用窗口函数 MAX OVER 和 MIN OVER 找出每个 DEPTNO 和 JOB 对应的最高和最低的工资。然后，筛选出工资与之匹配的行。

```
 1 select deptno,ename,job,sal,
 2        case when sal = max_by_dept
 3             then 'TOP SAL IN DEPT'
 4             when sal = min_by_dept
 5             then 'LOW SAL IN DEPT'
 6        end dept_status,
 7        case when sal = max_by_job
 8             then 'TOP SAL IN JOB'
 9             when sal = min_by_job
10             then 'LOW SAL IN JOB'
11        end job_status
12    from (
13  select deptno,ename,job,sal,
14         max(sal)over(partition by deptno) max_by_dept,
15         max(sal)over(partition by job)    max_by_job,
16         min(sal)over(partition by deptno) min_by_dept,
17         min(sal)over(partition by job)    min_by_job
18    from emp
19         ) emp_sals
20   where sal in (max_by_dept,max_by_job,
21                 min_by_dept,min_by_job)
```

PostgreSQL 和 MySQL

使用标量子查询找出每个 DEPTNO 和 JOB 对应的最高和最低的工资。然后，只保留与之匹配的员工。

```
 1 select deptno,ename,job,sal,
 2        case when sal = max_by_dept
 3             then 'TOP SAL IN DEPT'
 4             when sal = min_by_dept
 5             then 'LOW SAL IN DEPT'
 6        end as dept_status,
 7        case when sal = max_by_job
 8             then 'TOP SAL IN JOB'
 9             when sal = min_by_job
10             then 'LOW SAL IN JOB'
11        end as job_status
12   from (
```

```
13  select e.deptno,e.ename,e.job,e.sal,
14         (select max(sal) from  emp d
15           where d.deptno = e.deptno) as max_by_dept,
16         (select max(sal) from  emp d
17           where d.job = e.job) as max_by_job,
18         (select min(sal) from  emp d
19           where d.deptno = e.deptno) as min_by_dept,
20         (select min(sal) from  emp d
21           where d.job = e.job) as min_by_job
22    from emp e
23         ) x
24   where sal in (max_by_dept,max_by_job,
25                 min_by_dept,min_by_job)
```

3. 讨论

DB2、Oracle 和 SQL Server

首先使用窗口函数 MAX OVER 和 MIN OVER 找出每个 DEPTNO 和 JOB 对应的最高和最低的工资。

```
select deptno,ename,job,sal,
       max(sal)over(partition by deptno) maxDEPT,
       max(sal)over(partition by job)    maxJOB,
       min(sal)over(partition by deptno) minDEPT,
       min(sal)over(partition by job)    minJOB
  from emp

DEPTNO ENAME  JOB         SAL MAXDEPT MAXJOB MINDEPT MINJOB
------ ------ --------- ----- ------- ------ ------- ------
    10 MILLER CLERK      1300    5000   1300    1300    800
    10 CLARK  MANAGER    2450    5000   2975    1300   2450
    10 KING   PRESIDENT  5000    5000   5000    1300   5000
    20 SCOTT  ANALYST    3000    3000   3000     800   3000
    20 FORD   ANALYST    3000    3000   3000     800   3000
    20 SMITH  CLERK       800    3000   1300     800    800
    20 JONES  MANAGER    2975    3000   2975     800   2450
    20 ADAMS  CLERK      1100    3000   1300     800    800
    30 JAMES  CLERK       950    2850   1300     950    800
    30 MARTIN SALESMAN   1250    2850   1600     950   1250
    30 TURNER SALESMAN   1500    2850   1600     950   1250
    30 WARD   SALESMAN   1250    2850   1600     950   1250
    30 ALLEN  SALESMAN   1600    2850   1600     950   1250
    30 BLAKE  MANAGER    2850    2850   2975     950   2450
```

现在，每个人的工资都可以和当前 DEPTNO 和 JOB 对应的最高和最低的工资进行比较了。需要注意的是，上述窗口函数背后的分组操作（上述 SELECT 子句里的那 4 列）并不会影响 MIN OVER 和 MAX OVER 函数的返回值。这充分展现了窗口函数的优雅之处：聚合运算是基于指定的“分组”或分区展开的，并且会为每个分组返回多行数据。最后只要把上述窗口函数调用放入一个内嵌视图，并且只保留那些与窗口函数返回值相匹配的行即可。在最终的结果集中，我们将使用 CASE 表达式显示每个员工的“状态”。

```
select deptno,ename,job,sal,
       case when sal = max_by_dept
            then 'TOP SAL IN DEPT'
            when sal = min_by_dept
```

```
            then 'LOW SAL IN DEPT'
       end dept_status,
       case when sal = max_by_job
            then 'TOP SAL IN JOB'
            when sal = min_by_job
            then 'LOW SAL IN JOB'
       end job_status
  from (
select deptno,ename,job,sal,
       max(sal)over(partition by deptno) max_by_dept,
       max(sal)over(partition by job) max_by_job,
       min(sal)over(partition by deptno) min_by_dept,
       min(sal)over(partition by job) min_by_job
  from emp
       ) x
 where sal in (max_by_dept,max_by_job,
               min_by_dept,min_by_job)

DEPTNO ENAME  JOB         SAL DEPT_STATUS     JOB_STATUS
------ ------ --------- ----- --------------- --------------
    10 MILLER CLERK      1300 LOW SAL IN DEPT TOP SAL IN JOB
    10 CLARK  MANAGER    2450                 LOW SAL IN JOB
    10 KING   PRESIDENT  5000 TOP SAL IN DEPT TOP SAL IN JOB
    20 SCOTT  ANALYST    3000 TOP SAL IN DEPT TOP SAL IN JOB
    20 FORD   ANALYST    3000 TOP SAL IN DEPT TOP SAL IN JOB
    20 SMITH  CLERK       800 LOW SAL IN DEPT LOW SAL IN JOB
    20 JONES  MANAGER    2975                 TOP SAL IN JOB
    30 JAMES  CLERK       950 LOW SAL IN DEPT
    30 MARTIN SALESMAN   1250                 LOW SAL IN JOB
    30 WARD   SALESMAN   1250                 LOW SAL IN JOB
    30 ALLEN  SALESMAN   1600                 TOP SAL IN JOB
    30 BLAKE  MANAGER    2850 TOP SAL IN DEPT
```

PostgreSQL 和 MySQL

首先使用标量子查询找出每个 DEPTNO 和 JOB 对应的最高和最低的工资。

```
select e.deptno,e.ename,e.job,e.sal,
       (select max(sal) from  emp d
         where d.deptno = e.deptno) as maxDEPT,
       (select max(sal) from  emp d
         where d.job = e.job) as maxJOB,
       (select min(sal) from  emp d
         where d.deptno = e.deptno) as minDEPT,
       (select min(sal) from  emp d
         where d.job = e.job) as minJOB
  from emp e

DEPTNO ENAME  JOB         SAL MAXDEPT MAXJOB MINDEPT MINJOB
------ ------ --------- ----- ------- ------ ------- ------
    20 SMITH  CLERK       800    3000   1300     800    800
    30 ALLEN  SALESMAN   1600    2850   1600     950   1250
    30 WARD   SALESMAN   1250    2850   1600     950   1250
    20 JONES  MANAGER    2975    3000   2975     800   2450
    30 MARTIN SALESMAN   1250    2850   1600     950   1250
    30 BLAKE  MANAGER    2850    2850   2975     950   2450
```

```
   10 CLARK  MANAGER    2450    5000   2975    1300   2450
   20 SCOTT  ANALYST    3000    3000   3000     800   3000
   10 KING   PRESIDENT  5000    5000   5000    1300   5000
   30 TURNER SALESMAN   1500    2850   1600     950   1250
   20 ADAMS  CLERK      1100    3000   1300     800    800
   30 JAMES  CLERK       950    2850   1300     950    800
   20 FORD   ANALYST    3000    3000   3000     800   3000
   10 MILLER CLERK      1300    5000   1300    1300    800
```

现在，每个 DEPTNO 和 JOB 对应的最高和最低的工资可以和 EMP 表里其他的工资比较了。最后，把上述标量子查询放入到一个内嵌视图里，并且只保留那些工资与之相匹配的员工即可。在最终的结果集中，我们将使用 CASE 表达式显示每个员工的“状态”。

```
select deptno,ename,job,sal,
       case when sal = max_by_dept
            then 'TOP SAL IN DEPT'
            when sal = min_by_dept
            then 'LOW SAL IN DEPT'
       end as dept_status,
       case when sal = max_by_job
            then 'TOP SAL IN JOB'
            when sal = min_by_job
            then 'LOW SAL IN JOB'
       end as job_status
  from (
select e.deptno,e.ename,e.job,e.sal,
       (select max(sal) from  emp d
         where d.deptno = e.deptno) as max_by_dept,
       (select max(sal) from  emp d
         where d.job = e.job) as max_by_job,
       (select min(sal) from  emp d
         where d.deptno = e.deptno) as min_by_dept,
       (select min(sal) from  emp d
         where d.job = e.job) as min_by_job
  from emp e
       ) x
 where sal in (max_by_dept,max_by_job,
              min_by_dept,min_by_job)

DEPTNO ENAME  JOB         SAL DEPT_STATUS     JOB_STATUS
------ ------ --------- ----- --------------- --------------
    10 CLARK  MANAGER    2450                 LOW SAL IN JOB
    10 KING   PRESIDENT  5000 TOP SAL IN DEPT TOP SAL IN JOB
    10 MILLER CLERK      1300 LOW SAL IN DEPT TOP SAL IN JOB
    20 SMITH  CLERK       800 LOW SAL IN DEPT LOW SAL IN JOB
    20 FORD   ANALYST    3000 TOP SAL IN DEPT TOP SAL IN JOB
    20 SCOTT  ANALYST    3000 TOP SAL IN DEPT TOP SAL IN JOB
    20 JONES  MANAGER    2975                 TOP SAL IN JOB
    30 ALLEN  SALESMAN   1600                 TOP SAL IN JOB
    30 BLAKE  MANAGER    2850 TOP SAL IN DEPT
    30 MARTIN SALESMAN   1250                 LOW SAL IN JOB
    30 JAMES  CLERK       950 LOW SAL IN DEPT
    30 WARD   SALESMAN   1250                 LOW SAL IN JOB
```

12.12　计算简单的小计

1 .问题

在本实例中，“简单的小计”指的是一种特殊的结果集，该结果集不仅包括某一列的聚合运算结果，也包括了整个表中该列的合计值。例如，一个结果集里既包括了 EMP 表各个 JOB 对应的工资合计值，也包括了全部工资的总计。EMP 表各个 JOB 对应的工资合计值是小计，全部工资的合计值是总计。上述结果集看起来应该如下所示。

```
JOB              SAL
--------- ----------
ANALYST         6000
CLERK           4150
MANAGER         8275
PRESIDENT       5000
SALESMAN        5600
TOTAL          29025
```

2. 解决方案

针对 GROUP BY 子句的 ROLLUP 扩展完美地解决了本问题。对于不支持 ROLLUP 的数据库，可以借助标量子查询或 UNION 查询解决本问题，当然做法会繁琐一些。

DB2 和 Oracle

使用聚合函数 SUM 计算工资合计值，并使用 GROUP BY 的 ROLLUP 扩展构造出同时包含小计（按 JOB 分区）和总计（针对全表数据）的结果集。

```
1 select case grouping(job)
2             when 0 then job
3             else 'TOTAL'
4        end job,
5        sum(sal) sal
6   from emp
7  group by rollup(job)
```

SQL Server 和 MySQL

使用聚合函数 SUM 计算工资合计值，并使用 WITH ROLLUP 构造出同时包含小计（按 JOB 分区）和总计（针对全表数据）的结果集。然后调用 COALESCE 函数把总计行的标题改为 TOTAL（否则这一行的 JOB 列会出现 Null 值）。

```
1 select coalesce(job,'TOTAL') job,
2        sum(sal) sal
3   from emp
4  group by job with rollup
```

如果是 SQL Server，也可以不使用 COALESCE 函数，我们可以像上述 Oracle 和 DB2 的解决方案那样使用 GROUPING 函数来判断聚合运算的层级。

PostgreSQL

使用聚合函数 SUM 计算各个 DEPTNO 的工资合计值，然后使用 UNION ALL 把该查询和生成全表的工资总计的查询连在一起。

```
1 select job, sum(sal) as sal
2   from emp
3  group by job
4  union all
5 select 'TOTAL', sum(sal)
6   from emp
```

3. 讨论

DB2 和 Oracle

首先使用聚合函数 SUM，按照 JOB 分组并生成各个 JOB 的工资合计值。

```
select job, sum(sal) sal
  from emp
 group by job

JOB         SAL
--------- -----
ANALYST    6000
CLERK      4150
MANAGER    8275
PRESIDENT  5000
SALESMAN   5600
```

然后，使用 GROUP BY 的 ROLLUP 扩展在各个 JOB 的工资小计之外，再生成一个工资总计。

```
select job, sum(sal) sal
  from emp
 group by rollup(job)

JOB           SAL
--------- -------
ANALYST      6000
CLERK        4150
MANAGER      8275
PRESIDENT    5000
SALESMAN     5600
            29025
```

最后，借助 GROUPING 函数把工资总计行对应的 JOB 列的显示内容修改一下。如果 JOB 值是 Null，那么 GROUPING 函数会返回 1，这意味着 SAL 值是由 ROLLUP 生成的工资总计。如果 JOB 值不为 Null，则 GROUPING 函数将返回 0，这意味着 SAL 值是 GROUP BY 查询的结果，而不是 ROLLUP 的结果。在 CASE 表达式中调用 GROUPING(JOB)，这样就能根据需要返回具体的职位或代表总计行的标签 TOTAL。

```
select case grouping(job)
            when 0 then job
            else 'TOTAL'
        end job,
        sum(sal) sal
  from emp
 group by rollup(job)
```

```
JOB              SAL
--------- ----------
ANALYST         6000
CLERK           4150
MANAGER         8275
PRESIDENT       5000
SALESMAN        5600
TOTAL          29025
```

SQL Server 和 MySQL

首先使用聚合函数 SUM，按照 JOB 分组生成各个 JOB 的工资合计值。

```
select job, sum(sal) sal
  from emp
 group by job

JOB         SAL
--------- -----
ANALYST    6000
CLERK      4150
MANAGER    8275
PRESIDENT  5000
SALESMAN   5600
```

然后，使用 GROUP BY 的 ROLLUP 扩展在各个 JOB 的工资小计之外，再生成一个工资总计。

```
select job, sum(sal) sal
  from emp
 group by job with rollup

JOB           SAL
--------- -------
ANALYST      6000
CLERK        4150
MANAGER      8275
PRESIDENT    5000
SALESMAN     5600
            29025
```

最后，针对 JOB 列调用 COEALESCE 函数。如果 JOB 值为 Null，SAL 值就是由 ROLLUP 生成的工资总计。如果 JOB 值不为 Null，则 SAL 值是由通常的 GROUP BY 产生的结果，而不是 ROLLUP 的结果。

```
select coalesce(job,'TOTAL') job,
       sum(sal) sal
  from emp
 group by job with rollup

JOB              SAL
--------- ----------
ANALYST         6000
CLERK           4150
MANAGER         8275
PRESIDENT       5000
SALESMAN        5600
TOTAL          29025
```

PostgreSQL

首先对结果按照 JOB 分组，并使用聚合函数 SUM 生成各个 JOB 的工资合计值。

```
select job, sum(sal) sal
  from emp
 group by job

JOB         SAL
--------- -----
ANALYST    6000
CLERK      4150
MANAGER    8275
PRESIDENT  5000
SALESMAN   5600
```

最后，在上述查询的基础上使用 UNION ALL 查询生成的工资总计。

```
select job, sum(sal) as sal
  from emp
 group by job
 union all
select 'TOTAL', sum(sal)
  from emp

JOB           SAL
--------- -------
ANALYST      6000
CLERK        4150
MANAGER      8275
PRESIDENT    5000
SALESMAN     5600
TOTAL       29025
```

12.13　计算所有可能的表达式组合的小计

1. 问题

你想按照 DEPTNO、JOB 以及 JOB/DEPTNO 组合分别计算出工资合计值。同时，你也希望得到 EMP 表的工资总计。你希望得到如下所示的结果集。

```
DEPTNO JOB       CATEGORY                 SAL
------ --------- -------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB   1300
    10 MANAGER   TOTAL BY DEPT AND JOB   2450
    10 PRESIDENT TOTAL BY DEPT AND JOB   5000
    20 CLERK     TOTAL BY DEPT AND JOB   1900
    30 CLERK     TOTAL BY DEPT AND JOB    950
    30 SALESMAN  TOTAL BY DEPT AND JOB   5600
    30 MANAGER   TOTAL BY DEPT AND JOB   2850
    20 MANAGER   TOTAL BY DEPT AND JOB   2975
    20 ANALYST   TOTAL BY DEPT AND JOB   6000
       CLERK     TOTAL BY JOB            4150
       ANALYST   TOTAL BY JOB            6000
       MANAGER   TOTAL BY JOB            8275
```

```
   PRESIDENT TOTAL BY JOB            5000
   SALESMAN  TOTAL BY JOB            5600
10           TOTAL BY DEPT           8750
30           TOTAL BY DEPT           9400
20           TOTAL BY DEPT          10875
             GRAND TOTAL FOR TABLE  29025
```

2. 解决方案

近年来，对于 GROUP BY 语法的扩展使得本问题的解决变得容易多了。对于那些尚不支持这一类扩展语法的数据库，就必须（通过自连接或多个标量子查询）手动计算出多种层次的小计。

DB2

对于 DB2，需要把 GROUPING 函数的返回值转换为 CHAR(1) 类型。

```
 1 select deptno,
 2        job,
 3        case cast(grouping(deptno) as char(1))||
 4             cast(grouping(job) as char(1))
 5             when '00' then 'TOTAL BY DEPT AND JOB'
 6             when '10' then 'TOTAL BY JOB'
 7             when '01' then 'TOTAL BY DEPT'
 8             when '11' then 'TOTAL FOR TABLE'
 9        end category,
10        sum(sal)
11   from emp
12  group by cube(deptno,job)
13  order by grouping(job),grouping(deptno)
```

Oracle

使用 GROUP BY 子句的 CUBE 扩展以及字符串连接操作符 ||。

```
 1 select deptno,
 2        job,
 3        case grouping(deptno)||grouping(job)
 4             when '00' then 'TOTAL BY DEPT AND JOB'
 5             when '10' then 'TOTAL BY JOB'
 6             when '01' then 'TOTAL BY DEPT'
 7             when '11' then 'GRAND TOTAL FOR TABLE'
 8        end category,
 9        sum(sal) sal
10   from emp
11  group by cube(deptno,job)
12  order by grouping(job),grouping(deptno)
```

SQL Server

使用 GROUP BY 子句的 CUBE 扩展。对于 SQL Server，需要把 GROUPING 函数的返回值转换成 CHAR(1) 类型，并且要使用字符串连接操作符 +（不同于 Oracle 的 || 操作符）。

```
 1 select deptno,
 2        job,
 3        case cast(grouping(deptno)as char(1))+
 4             cast(grouping(job)as char(1))
```

```
 5               when '00' then 'TOTAL BY DEPT AND JOB'
 6               when '10' then 'TOTAL BY JOB'
 7               when '01' then 'TOTAL BY DEPT'
 8               when '11' then 'GRAND TOTAL FOR TABLE'
 9          end category,
10          sum(sal) sal
11     from emp
12    group by deptno,job with cube
13    order by grouping(job),grouping(deptno)
```

PostgreSQL 和 MySQL

使用多个 UNION ALL，把每种类型的合计合并到一起。

```
 1  select deptno, job,
 2         'TOTAL BY DEPT AND JOB' as category,
 3         sum(sal) as sal
 4    from emp
 5   group by deptno, job
 6   union all
 7  select null, job, 'TOTAL BY JOB', sum(sal)
 8    from emp
 9   group by job
10  union all
11 select deptno, null, 'TOTAL BY DEPT', sum(sal)
12   from emp
13  group by deptno
14  union all
15 select null,null,'GRAND TOTAL FOR TABLE', sum(sal)
16   from emp
```

3. 讨论

Oracle、DB2 和 SQL Server

这 3 种数据库的解决方案大体相同。首先使用聚合函数 SUM，按照 DEPTNO 和 JOB 分组，找出每个 JOB 和 DEPTNO 组合对应的工资合计值。

```
select deptno, job, sum(sal) sal
  from emp
 group by deptno, job

DEPTNO JOB           SAL
------ --------- -------
    10 CLERK        1300
    10 MANAGER      2450
    10 PRESIDENT    5000
    20 CLERK        1900
    20 ANALYST      6000
    20 MANAGER      2975
    30 CLERK         950
    30 MANAGER      2850
    30 SALESMAN     5600
```

下一步是生成 JOB 和 DEPTNO 的工资小计，以及全表的工资总计。使用 GROUP BY 子句的 CUBE 扩展，分别按照 DEPTNO、JOB 和全表的维度执行 SAL 列的聚合运算。

```
select deptno,
       job,
       sum(sal) sal
  from emp
 group by cube(deptno,job)

DEPTNO JOB           SAL
------ --------- -------
                   29025
       CLERK        4150
       ANALYST      6000
       MANAGER      8275
       SALESMAN     5600
       PRESIDENT    5000
    10              8750
    10 CLERK        1300
    10 MANAGER      2450
    10 PRESIDENT    5000
    20             10875
    20 CLERK        1900
    20 ANALYST      6000
    20 MANAGER      2975
    30              9400
    30 CLERK         950
    30 MANAGER      2850
    30 SALESMAN     5600
```

然后，使用 GROUPING 函数和 CASE 表达式把上述结果格式化为更有意义的输出。GROUPING(JOB) 的返回值应该是 1 或 0，这取决于 SAL 值是否来自 CUBE。如果结果来自 CUBE，则返回值为 0，否则返回 1。GROUPING(DEPTNO) 的返回值也是如此。再回头看一下本解决方案的第一步，我们知道它是按照 DEPTNO 和 JOB 进行分组的。因此，如果 SAL 值是基于 DEPTNO 和 JOB 组合计算而来的，那么 GROUPING 函数调用的返回值将是 0。执行结果显示如下。

```
select deptno,
       job,
       grouping(deptno) is_deptno_subtotal,
       grouping(job) is_job_subtotal,
       sum(sal) sal
  from emp
 group by cube(deptno,job)
 order by 3,4

DEPTNO JOB       IS_DEPTNO_SUBTOTAL IS_JOB_SUBTOTAL     SAL
------ --------- ------------------ --------------- -------
    10 CLERK                      0               0    1300
    10 MANAGER                    0               0    2450
    10 PRESIDENT                  0               0    5000
    20 CLERK                      0               0    1900
    30 CLERK                      0               0     950
    30 SALESMAN                   0               0    5600
    30 MANAGER                    0               0    2850
    20 MANAGER                    0               0    2975
    20 ANALYST                    0               0    6000
    10                            0               1    8750
```

```
20                         0              1   10875
30                         0              1    9400
   CLERK                   1              0    4150
   ANALYST                 1              0    6000
   MANAGER                 1              0    8275
   PRESIDENT               1              0    5000
   SALESMAN                1              0    5600
                           1              1   29025
```

最后，使用 CASE 表达式确认每一行的归属，这是基于 GROUPING(JOB) 和 GROUPING(DEPTNO) 的返回值来决定的。

```
select deptno,
       job,
       case grouping(deptno)||grouping(job)
            when '00' then 'TOTAL BY DEPT AND JOB'
            when '10' then 'TOTAL BY JOB'
            when '01' then 'TOTAL BY DEPT'
            when '11' then 'GRAND TOTAL FOR TABLE'
       end category,
       sum(sal) sal
  from emp
 group by cube(deptno,job)
 order by grouping(job),grouping(deptno)

DEPTNO JOB       CATEGORY                  SAL
------ --------- --------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB    1300
    10 MANAGER   TOTAL BY DEPT AND JOB    2450
    10 PRESIDENT TOTAL BY DEPT AND JOB    5000
    20 CLERK     TOTAL BY DEPT AND JOB    1900
    30 CLERK     TOTAL BY DEPT AND JOB     950
    30 SALESMAN  TOTAL BY DEPT AND JOB    5600
    30 MANAGER   TOTAL BY DEPT AND JOB    2850
    20 MANAGER   TOTAL BY DEPT AND JOB    2975
    20 ANALYST   TOTAL BY DEPT AND JOB    6000
       CLERK     TOTAL BY JOB             4150
       ANALYST   TOTAL BY JOB             6000
       MANAGER   TOTAL BY JOB             8275
       PRESIDENT TOTAL BY JOB             5000
       SALESMAN  TOTAL BY JOB             5600
    10           TOTAL BY DEPT            8750
    30           TOTAL BY DEPT            9400
    20           TOTAL BY DEPT           10875
                 GRAND TOTAL FOR TABLE   29025
```

上述 Oracle 解决方案在做字符串连接操作时，把两个 GROUPING 函数的返回值隐式地转换成了字符类型。对于 DB2 和 SQL Server 而言，则需要显式地把 GROUPING 函数的返回值转换成 CHAR(1) 数据类型，正如上述解决方案所示。另外，把两个 GROUPING 函数调用的返回值拼接成一个字符串时，SQL Server 用户必须使用 + 操作符，而不是 || 操作符。

对于 Oracle 和 DB2 而言，GROUP BY 还有一个 GROUPING SETS 语法扩展，该扩展也非常有用。例如，我们可以用 GROUPING SETS 模仿 CUBE 的输出结果，如下所示。（和 CUBE 解决方案一

样，DB2 和 SQL Server 需要对 GROUPING 函数的返回值进行显式的数据类型转换。）

```
select deptno,
       job,
       case grouping(deptno)||grouping(job)
            when '00' then 'TOTAL BY DEPT AND JOB'
            when '10' then 'TOTAL BY JOB'
            when '01' then 'TOTAL BY DEPT'
            when '11' then 'GRAND TOTAL FOR TABLE'
       end category,
       sum(sal) sal
  from emp
 group by grouping sets ((deptno),(job),(deptno,job),())

DEPTNO JOB       CATEGORY                  SAL
------ --------- --------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB    1300
    20 CLERK     TOTAL BY DEPT AND JOB    1900
    30 CLERK     TOTAL BY DEPT AND JOB     950
    20 ANALYST   TOTAL BY DEPT AND JOB    6000
    10 MANAGER   TOTAL BY DEPT AND JOB    2450
    20 MANAGER   TOTAL BY DEPT AND JOB    2975
    30 MANAGER   TOTAL BY DEPT AND JOB    2850
    30 SALESMAN  TOTAL BY DEPT AND JOB    5600
    10 PRESIDENT TOTAL BY DEPT AND JOB    5000
       CLERK     TOTAL BY JOB             4150
       ANALYST   TOTAL BY JOB             6000
       MANAGER   TOTAL BY JOB             8275
       SALESMAN  TOTAL BY JOB             5600
       PRESIDENT TOTAL BY JOB             5000
    10           TOTAL BY DEPT            8750
    20           TOTAL BY DEPT           10875
    30           TOTAL BY DEPT            9400
                 GRAND TOTAL FOR TABLE   29025
```

GROUPING SETS 的奇妙之处在于它允许我们定义分组。上述查询中的 GROUPING SETS 子句分别指定了按照 DEPTNO 分组，按照 JOB 分组，按照 DEPTNO 和 JOB 的组合分组，以及最后空白括号代表的总计。GROUPING SETS 能够非常灵活地支持不同维度的聚合运算。例如，如果我们希望改一下上面的例子，去掉 GRAND TOTAL，那么只要把 GROUPING SETS 子句的空白括号删除掉就可以了。

```
/*去掉总计 */

select deptno,
       job,
       case grouping(deptno)||grouping(job)
            when '00' then 'TOTAL BY DEPT AND JOB'
            when '10' then 'TOTAL BY JOB'
            when '01' then 'TOTAL BY DEPT'
            when '11' then 'GRAND TOTAL FOR TABLE'
       end category,
       sum(sal) sal
  from emp
 group by grouping sets ((deptno),(job),(deptno,job))
```

```
DEPTNO JOB       CATEGORY                    SAL
------ --------- -------------------- ----------
    10 CLERK     TOTAL BY DEPT AND JOB      1300
    20 CLERK     TOTAL BY DEPT AND JOB      1900
    30 CLERK     TOTAL BY DEPT AND JOB       950
    20 ANALYST   TOTAL BY DEPT AND JOB      6000
    10 MANAGER   TOTAL BY DEPT AND JOB      2450
    20 MANAGER   TOTAL BY DEPT AND JOB      2975
    30 MANAGER   TOTAL BY DEPT AND JOB      2850
    30 SALESMAN  TOTAL BY DEPT AND JOB      5600
    10 PRESIDENT TOTAL BY DEPT AND JOB      5000
       CLERK     TOTAL BY JOB               4150
       ANALYST   TOTAL BY JOB               6000
       MANAGER   TOTAL BY JOB               8275
       SALESMAN  TOTAL BY JOB               5600
       PRESIDENT TOTAL BY JOB               5000
    10           TOTAL BY DEPT              8750
    20           TOTAL BY DEPT             10875
    30           TOTAL BY DEPT              9400
```

我们也可以删除一个小计，例如基于 DEPTNO 的小计，只要从 GROUPING SETS 子句中去掉“(DEPTNO)”就可以了。

```
/* 去掉DEPTNO小计 */

select deptno,
       job,
       case grouping(deptno)||grouping(job)
            when '00' then 'TOTAL BY DEPT AND JOB'
            when '10' then 'TOTAL BY JOB'
            when '01' then 'TOTAL BY DEPT'
            when '11' then 'GRAND TOTAL FOR TABLE'
       end category,
       sum(sal) sal
  from emp
 group by grouping sets ((job),(deptno,job),())
 order by 3

DEPTNO JOB       CATEGORY                    SAL
------ --------- -------------------- ----------
                 GRAND TOTAL FOR TABLE     29025
    10 CLERK     TOTAL BY DEPT AND JOB      1300
    20 CLERK     TOTAL BY DEPT AND JOB      1900
    30 CLERK     TOTAL BY DEPT AND JOB       950
    20 ANALYST   TOTAL BY DEPT AND JOB      6000
    20 MANAGER   TOTAL BY DEPT AND JOB      2975
    30 MANAGER   TOTAL BY DEPT AND JOB      2850
    30 SALESMAN  TOTAL BY DEPT AND JOB      5600
    10 PRESIDENT TOTAL BY DEPT AND JOB      5000
    10 MANAGER   TOTAL BY DEPT AND JOB      2450
       CLERK     TOTAL BY JOB               4150
       SALESMAN  TOTAL BY JOB               5600
       PRESIDENT TOTAL BY JOB               5000
       MANAGER   TOTAL BY JOB               8275
       ANALYST   TOTAL BY JOB               6000
```

如上所述，GROUPING SETS 确实能更方便地帮助我们从不同角度获得总计和小计。

PostgreSQL 和 MySQL

首先使用聚合函数 SUM，并按照 DEPTNO 和 JOB 进行分组。

```
select deptno, job,
       'TOTAL BY DEPT AND JOB' as category,
       sum(sal) as sal
  from emp
 group by deptno, job

DEPTNO JOB       CATEGORY                  SAL
------ --------- --------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB    1300
    10 MANAGER   TOTAL BY DEPT AND JOB    2450
    10 PRESIDENT TOTAL BY DEPT AND JOB    5000
    20 CLERK     TOTAL BY DEPT AND JOB    1900
    20 ANALYST   TOTAL BY DEPT AND JOB    6000
    20 MANAGER   TOTAL BY DEPT AND JOB    2975
    30 CLERK     TOTAL BY DEPT AND JOB     950
    30 MANAGER   TOTAL BY DEPT AND JOB    2850
    30 SALESMAN  TOTAL BY DEPT AND JOB    5600
```

然后，使用 UNION ALL 把基于 JOB 分组的工资合计值合并进来。

```
select deptno, job,
       'TOTAL BY DEPT AND JOB' as category,
       sum(sal) as sal
  from emp
 group by deptno, job
 union all
select null, job, 'TOTAL BY JOB', sum(sal)
  from emp
 group by job

DEPTNO JOB       CATEGORY                  SAL
------ --------- --------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB    1300
    10 MANAGER   TOTAL BY DEPT AND JOB    2450
    10 PRESIDENT TOTAL BY DEPT AND JOB    5000
    20 CLERK     TOTAL BY DEPT AND JOB    1900
    20 ANALYST   TOTAL BY DEPT AND JOB    6000
    20 MANAGER   TOTAL BY DEPT AND JOB    2975
    30 CLERK     TOTAL BY DEPT AND JOB     950
    30 MANAGER   TOTAL BY DEPT AND JOB    2850
    30 SALESMAN  TOTAL BY DEPT AND JOB    5600
       ANALYST   TOTAL BY JOB             6000
       CLERK     TOTAL BY JOB             4150
       MANAGER   TOTAL BY JOB             8275
       PRESIDENT TOTAL BY JOB             5000
       SALESMAN  TOTAL BY JOB             5600
```

再使用 UNION ALL 把基于 DEPTNO 分组的工资合计值合并进来。

```
select deptno, job,
       'TOTAL BY DEPT AND JOB' as category,
       sum(sal) as sal
```

```
  from emp
 group by deptno, job
 union all
select null, job, 'TOTAL BY JOB', sum(sal)
  from emp
 group by job
 union all
select deptno, null, 'TOTAL BY DEPT', sum(sal)
  from emp
 group by deptno

DEPTNO JOB       CATEGORY                 SAL
------ --------- -------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB    1300
    10 MANAGER   TOTAL BY DEPT AND JOB    2450
    10 PRESIDENT TOTAL BY DEPT AND JOB    5000
    20 CLERK     TOTAL BY DEPT AND JOB    1900
    20 ANALYST   TOTAL BY DEPT AND JOB    6000
    20 MANAGER   TOTAL BY DEPT AND JOB    2975
    30 CLERK     TOTAL BY DEPT AND JOB     950
    30 MANAGER   TOTAL BY DEPT AND JOB    2850
    30 SALESMAN  TOTAL BY DEPT AND JOB    5600
       ANALYST   TOTAL BY JOB             6000
       CLERK     TOTAL BY JOB             4150
       MANAGER   TOTAL BY JOB             8275
       PRESIDENT TOTAL BY JOB             5000
       SALESMAN  TOTAL BY JOB             5600
    10           TOTAL BY DEPT            8750
    20           TOTAL BY DEPT           10875
    30           TOTAL BY DEPT            9400
```

最后，使用 UNION ALL 把 EMP 表的工资总计合并进来。

```
select deptno, job,
       'TOTAL BY DEPT AND JOB' as category,
       sum(sal) as sal
  from emp
 group by deptno, job
 union all
select null, job, 'TOTAL BY JOB', sum(sal)
  from emp
 group by job
 union all
select deptno, null, 'TOTAL BY DEPT', sum(sal)
  from emp
 group by deptno
 union all
select null,null, 'GRAND TOTAL FOR TABLE', sum(sal)
  from emp

DEPTNO JOB       CATEGORY                 SAL
------ --------- -------------------- -------
    10 CLERK     TOTAL BY DEPT AND JOB    1300
    10 MANAGER   TOTAL BY DEPT AND JOB    2450
    10 PRESIDENT TOTAL BY DEPT AND JOB    5000
```

```
20 CLERK     TOTAL BY DEPT AND JOB    1900
20 ANALYST   TOTAL BY DEPT AND JOB    6000
20 MANAGER   TOTAL BY DEPT AND JOB    2975
30 CLERK     TOTAL BY DEPT AND JOB     950
30 MANAGER   TOTAL BY DEPT AND JOB    2850
30 SALESMAN  TOTAL BY DEPT AND JOB    5600
   ANALYST   TOTAL BY JOB             6000
   CLERK     TOTAL BY JOB             4150
   MANAGER   TOTAL BY JOB             8275
   PRESIDENT TOTAL BY JOB             5000
   SALESMAN  TOTAL BY JOB             5600
10           TOTAL BY DEPT            8750
20           TOTAL BY DEPT           10875
30           TOTAL BY DEPT            9400
             GRAND TOTAL FOR TABLE   29025
```

12.14 识别非小计行

1. 问题

你已经知道如何使用 GROUP BY 子句的 CUBE 扩展语法生成报表，并且你需要知道如何区分哪些行是由普通的 GROUP BY 子句产生的，哪些行是由 CUBE 或 ROLLUP 产生的。

下面给出了一组利用 GROUP BY 的 CUBE 扩展语法生成的查询结果集，它是 EMP 表中员工工资的一个分类汇总结果。

```
DEPTNO JOB           SAL
------ --------- -------
                   29025
       CLERK        4150
       ANALYST      6000
       MANAGER      8275
       SALESMAN     5600
       PRESIDENT    5000
    10              8750
    10 CLERK        1300
    10 MANAGER      2450
    10 PRESIDENT    5000
    20             10875
    20 CLERK        1900
    20 ANALYST      6000
    20 MANAGER      2975
    30              9400
    30 CLERK         950
    30 MANAGER      2850
    30 SALESMAN     5600
```

以上报表展示了按照 DEPTNO 和 JOB 分组计算出来的（每个 DEPTNO 对应的每一种 JOB 的）工资合计值，按照 DEPTNO 分组计算出来的工资合计值，按照 JOB 分组计算出来的工资合计值，以及最后的总计（EMP 表的工资合计值）。你希望能清楚地识别上述各种聚合运算的维度，并标记出每一个聚合运算结果分别属于哪一类。（也就是说，给定的 SAL 值代表的是按照 DEPTNO 分组计算的结果，还是按照 JOB 分组计算的结果，或者是总计？）你希望返回

如下所示的结果集。

```
DEPTNO JOB           SAL DEPTNO_SUBTOTALS JOB_SUBTOTALS
------ --------- ------- ---------------- -------------
               29025                1             1
       CLERK     4150                1             0
       ANALYST   6000                1             0
       MANAGER   8275                1             0
       SALESMAN  5600                1             0
       PRESIDENT 5000                1             0
    10           8750                0             1
    10 CLERK     1300                0             0
    10 MANAGER   2450                0             0
    10 PRESIDENT 5000                0             0
    20          10875                0             1
    20 CLERK     1900                0             0
    20 ANALYST   6000                0             0
    20 MANAGER   2975                0             0
    30           9400                0             1
    30 CLERK      950                0             0
    30 MANAGER   2850                0             0
    30 SALESMAN  5600                0             0
```

2. 解决方案

使用 GROUPING 函数判断哪些值是 CUBE 或 ROLLUP 的小计结果，即**超级聚合**（supera ggregate）值。下面是 DB2 和 Oracle 的示例代码。

```
1  select deptno, job, sum(sal) sal,
2         grouping(deptno) deptno_subtotals,
3         grouping(job) job_subtotals
4    from emp
5   group by cube(deptno,job)
```

相较于 DB2 和 Oracle 解决方案，SQL Server 解决方案唯一的不同之处在于 CUBE/ROLLUP 子句的语法。

```
1  select deptno, job, sum(sal) sal,
2         grouping(deptno) deptno_subtotals,
3         grouping(job) job_subtotals
4    from emp
5   group by deptno,job with cube
```

本实例的重点在于展示如何使用 CUBE 和 GROUPING 处理小计计算。在写作本书时，PostgreSQL 和 MySQL 尚不支持 CUBE 或 GROUPING。

3. 讨论

如果 DEPTNO_SUBTOTALS 等于 0，并且 JOB_SUBTOTALS 等于 1（此时 JOB 是 Null），那么 SAL 值就是 CUBE 查询生成的、按照 DEPTNO 分组的小计结果。如果 JOB_SUBTOTALS 等于 0，并且 DEPTNO_SUBTOTALS 等于 1（此时 DEPTNO 是 Null），那么 SAL 值就是 CUBE 查询生成的、按照 JOB 分组的小计结果。如果 JOB_SUBTOTALS 和 DEPTNO_ SUBTOTALS 都等于 1，那么 SAL 值就是 CUBE 查询生成的工资总计。DEPTNO_SUBTOTALS 和 JOB_SUBTOTALS 都等于 0 的行则是通常的聚合运算结果（每个 DEPTNO/JOB 组合对应的 SAL 合计）。

12.15 使用CASE表达式标记行数据

1. 问题

你想把某列的值映射成一系列的“布尔”标志位。例如，对于 EMP 表的 JOB 列，你希望得到如下所示的结果集。

```
ENAME  IS_CLERK IS_SALES IS_MGR IS_ANALYST IS_PREZ
------ -------- -------- ------ ---------- -------
KING          0        0      0          0       1
SCOTT         0        0      0          1       0
FORD          0        0      0          1       0
JONES         0        0      1          0       0
BLAKE         0        0      1          0       0
CLARK         0        0      1          0       0
ALLEN         0        1      0          0       0
WARD          0        1      0          0       0
MARTIN        0        1      0          0       0
TURNER        0        1      0          0       0
SMITH         1        0      0          0       0
MILLER        1        0      0          0       0
ADAMS         1        0      0          0       0
JAMES         1        0      0          0       0
```

类似上述这样的结果集在调试程序的时候往往很有用，并且它也提供了一种不同于普通结果集的数据视图。

2. 解决方案

使用 CASE 表达式评估每个员工的 JOB 值，并返回 1 或 0 以标记评估结果。我们需要写一组 CASE 表达式为每一种可能的 JOB 值创建一列返回值。

```
 1 select ename,
 2        case when job = 'CLERK'
 3             then 1 else 0
 4        end as is_clerk,
 5        case when job = 'SALESMAN'
 6             then 1 else 0
 7        end as is_sales,
 8        case when job = 'MANAGER'
 9             then 1 else 0
10        end as is_mgr,
11        case when job = 'ANALYST'
12             then 1 else 0
13        end as is_analyst,
14        case when job = 'PRESIDENT'
15             then 1 else 0
16        end as is_prez
17   from emp
18  order by 2,3,4,5,6
```

3. 讨论

本解决方案的代码非常易于理解。如果你还是不太理解的话，不妨把 JOB 列也放入到 SELECT 子句里。

```
select ename,
       job,
       case when job = 'CLERK'
            then 1 else 0
       end as is_clerk,
       case when job = 'SALESMAN'
            then 1 else 0
       end as is_sales,
       case when job = 'MANAGER'
            then 1 else 0
       end as is_mgr,
       case when job = 'ANALYST'
            then 1 else 0
       end as is_analyst,
       case when job = 'PRESIDENT'
            then 1 else 0
       end as is_prez
  from emp
 order by 2
```

```
ENAME  JOB       IS_CLERK IS_SALES IS_MGR IS_ANALYST IS_PREZ
------ --------- -------- -------- ------ ---------- -------
SCOTT  ANALYST          0        0      0          1       0
FORD   ANALYST          0        0      0          1       0
SMITH  CLERK            1        0      0          0       0
ADAMS  CLERK            1        0      0          0       0
MILLER CLERK            1        0      0          0       0
JAMES  CLERK            1        0      0          0       0
JONES  MANAGER          0        0      1          0       0
CLARK  MANAGER          0        0      1          0       0
BLAKE  MANAGER          0        0      1          0       0
KING   PRESIDENT        0        0      0          0       1
ALLEN  SALESMAN         0        1      0          0       0
MARTIN SALESMAN         0        1      0          0       0
TURNER SALESMAN         0        1      0          0       0
WARD   SALESMAN         0        1      0          0       0
```

12.16 创建稀疏矩阵

1. 问题

你想创建一个稀疏矩阵，把 EMP 表的 DEPTNO 和 JOB 列变换成如下所示的结果集。

```
D10        D20        D30        CLERKS MGRS  PREZ ANALS SALES
---------- ---------- ---------- ------ ----- ---- ----- ------
           SMITH                 SMITH
                      ALLEN                                ALLEN
                      WARD                                 WARD
           JONES                        JONES
                      MARTIN                               MARTIN
                      BLAKE             BLAKE
CLARK                                   CLARK
           SCOTT                                     SCOTT
KING                                          KING
```

```
                      TURNER                             TURNER
           ADAMS                 ADAMS
                      JAMES      JAMES
           FORD                                   FORD
MILLER                           MILLER
```

2. 解决方案

使用一组 CASE 表达式创建一个稀有矩阵，实现从行到列的变换。

```
1  select case deptno when 10 then ename end as d10,
2         case deptno when 20 then ename end as d20,
3         case deptno when 30 then ename end as d30,
4         case job when 'CLERK'     then ename end as clerks,
5         case job when 'MANAGER'   then ename end as mgrs,
6         case job when 'PRESIDENT' then ename end as prez,
7         case job when 'ANALYST'   then ename end as anals,
8         case job when 'SALESMAN'  then ename end as sales
9    from emp
```

3. 讨论

要把 DEPTNO 和 JOB 值从行形式变换成列形式，只要使用 CASE 表达式把全部可能的值逐一评估一遍即可。这是本解决方案的要点所在。除此之外，如果希望删除一些 Null 行，以便让整个报表显得“紧密”一些，我们还需要按照某个列做分组操作。例如，使用窗口函数 ROW_NUMBER OVER 为每个 DEPTNO 对应的每个员工生成一个序号，然后使用聚合函数 MAX 删除一些 Null 值。

```
select max(case deptno when 10 then ename end) d10,
       max(case deptno when 20 then ename end) d20,
       max(case deptno when 30 then ename end) d30,
       max(case job when 'CLERK'     then ename end) clerks,
       max(case job when 'MANAGER'   then ename end) mgrs,
       max(case job when 'PRESIDENT' then ename end) prez,
       max(case job when 'ANALYST'   then ename end) anals,
       max(case job when 'SALESMAN'  then ename end) sales
  from (
select deptno, job, ename,
       row_number()over(partition by deptno order by empno) rn
  from emp
       ) x
 group by rn

D10        D20        D30        CLERKS MGRS  PREZ ANALS SALES
---------- ---------- ---------- ------ ----- ---- ----- ------
CLARK      SMITH      ALLEN      SMITH  CLARK            ALLEN
KING       JONES      WARD              JONES KING       WARD
MILLER     SCOTT      MARTIN     MILLER            SCOTT MARTIN
           ADAMS      BLAKE      ADAMS  BLAKE
           FORD       TURNER                       FORD  TURNER
                      JAMES      JAMES
```

12.17 按照时间单位分组

1. 问题

你想按照某种时间间隔整理数据。例如，你有一些交易日志，并希望每隔 5 秒汇总一下这些数据。TRX_LOG 表的数据显示如下。

```
select trx_id,
       trx_date,
       trx_cnt
  from trx_log

TRX_ID TRX_DATE                TRX_CNT
------ -------------------- ----------
     1 28-JUL-2005 19:03:07         44
     2 28-JUL-2005 19:03:08         18
     3 28-JUL-2005 19:03:09         23
     4 28-JUL-2005 19:03:10         29
     5 28-JUL-2005 19:03:11         27
     6 28-JUL-2005 19:03:12         45
     7 28-JUL-2005 19:03:13         45
     8 28-JUL-2005 19:03:14         32
     9 28-JUL-2005 19:03:15         41
    10 28-JUL-2005 19:03:16         15
    11 28-JUL-2005 19:03:17         24
    12 28-JUL-2005 19:03:18         47
    13 28-JUL-2005 19:03:19         37
    14 28-JUL-2005 19:03:20         48
    15 28-JUL-2005 19:03:21         46
    16 28-JUL-2005 19:03:22         44
    17 28-JUL-2005 19:03:23         36
    18 28-JUL-2005 19:03:24         41
    19 28-JUL-2005 19:03:25         33
    20 28-JUL-2005 19:03:26         19
```

你希望返回如下所示的结果集。

```
GRP TRX_START            TRX_END                   TOTAL
--- -------------------- -------------------- ----------
  1 28-JUL-2005 19:03:07 28-JUL-2005 19:03:11        141
  2 28-JUL-2005 19:03:12 28-JUL-2005 19:03:16        178
  3 28-JUL-2005 19:03:17 28-JUL-2005 19:03:21        202
  4 28-JUL-2005 19:03:22 28-JUL-2005 19:03:26        173
```

2. 解决方案

把全部数据记录分别放入若干个桶，每个桶里放 5 行。要实现这种逻辑分组，有几种可能的实现方式。本实例的做法是用 TRX_ID 值除以 5，12.7 节曾使用过该技巧。

创建好“分组”之后，调用聚合函数 MIN、MAX 和 SUM 分别计算出开始时间、结束时间和每个“分组”的交易数目合计（如果是 SQL Server 的话，要记得用 CEILING 函数替换 CEIL）。

```
1  select ceil(trx_id/5.0) as grp,
2         min(trx_date)    as trx_start,
3         max(trx_date)    as trx_end,
```

```
4         sum(trx_cnt)     as total
5    from trx_log
6   group by ceil(trx_id/5.0)
```

3. 讨论

首先要对所有行数据进行逻辑分组，这是整个解决方案的关键所在。TRX_ID 除以 5，并找到大于该商值的最小整数，这样我们就实现了逻辑分组。例如：

```
select trx_id,
       trx_date,
       trx_cnt,
       trx_id/5.0       as val,
       ceil(trx_id/5.0) as grp
  from trx_log

TRX_ID TRX_DATE             TRX_CNT    VAL GRP
------ -------------------- ------- ------ ---
     1 28-JUL-2005 19:03:07      44    .20   1
     2 28-JUL-2005 19:03:08      18    .40   1
     3 28-JUL-2005 19:03:09      23    .60   1
     4 28-JUL-2005 19:03:10      29    .80   1
     5 28-JUL-2005 19:03:11      27   1.00   1
     6 28-JUL-2005 19:03:12      45   1.20   2
     7 28-JUL-2005 19:03:13      45   1.40   2
     8 28-JUL-2005 19:03:14      32   1.60   2
     9 28-JUL-2005 19:03:15      41   1.80   2
    10 28-JUL-2005 19:03:16      15   2.00   2
    11 28-JUL-2005 19:03:17      24   2.20   3
    12 28-JUL-2005 19:03:18      47   2.40   3
    13 28-JUL-2005 19:03:19      37   2.60   3
    14 28-JUL-2005 19:03:20      48   2.80   3
    15 28-JUL-2005 19:03:21      46   3.00   3
    16 28-JUL-2005 19:03:22      44   3.20   4
    17 28-JUL-2005 19:03:23      36   3.40   4
    18 28-JUL-2005 19:03:24      41   3.60   4
    19 28-JUL-2005 19:03:25      33   3.80   4
    20 28-JUL-2005 19:03:26      19   4.00   4
```

最后，调用合适的聚合函数计算出每 5 秒钟有多少个交易，同时找出每一组交易的开始时间和结束时间。

```
select ceil(trx_id/5.0) as grp,
       min(trx_date)    as trx_start,
       max(trx_date)    as trx_end,
       sum(trx_cnt)     as total
  from trx_log
 group by ceil(trx_id/5.0)

GRP TRX_START            TRX_END                   TOTAL
--- -------------------- -------------------- ----------
  1 28-JUL-2005 19:03:07 28-JUL-2005 19:03:11        141
  2 28-JUL-2005 19:03:12 28-JUL-2005 19:03:16        178
  3 28-JUL-2005 19:03:17 28-JUL-2005 19:03:21        202
  4 28-JUL-2005 19:03:22 28-JUL-2005 19:03:26        173
```

如果输入数据的格式与 TRX_LOG 表有所不同（例如缺少了 ID 列），我们可以用每一行的 TRX_DATE 对应的“秒”值除以 5，这样也能生成和上述解决方案类似的分组。然后，我们可以把每个 TRX_DATE 对应的“小时”也一并提取出来，并按照小时和“逻辑分组”GRP 值进行分组。下面的示例展示了这一技巧（此处用到了 Oracle 的 TO_CHAR 和 TO_NUMBER 函数，对于其他数据库需要使用适当的日期和字符串格式函数）。

```
select trx_date,trx_cnt,
       to_number(to_char(trx_date,'hh24')) hr,
       ceil(to_number(to_char(trx_date-1/24/60/60,'miss'))/5.0) grp
  from trx_log

TRX_DATE                TRX_CNT         HR        GRP
-------------------- ---------- ---------- ----------
28-JUL-2005 19:03:07         44         19         62
28-JUL-2005 19:03:08         18         19         62
28-JUL-2005 19:03:09         23         19         62
28-JUL-2005 19:03:10         29         19         62
28-JUL-2005 19:03:11         27         19         62
28-JUL-2005 19:03:12         45         19         63
28-JUL-2005 19:03:13         45         19         63
28-JUL-2005 19:03:14         32         19         63
28-JUL-2005 19:03:15         41         19         63
28-JUL-2005 19:03:16         15         19         63
28-JUL-2005 19:03:17         24         19         64
28-JUL-2005 19:03:18         47         19         64
28-JUL-2005 19:03:19         37         19         64
28-JUL-2005 19:03:20         48         19         64
28-JUL-2005 19:03:21         46         19         64
28-JUL-2005 19:03:22         44         19         65
28-JUL-2005 19:03:23         36         19         65
28-JUL-2005 19:03:24         41         19         65
28-JUL-2005 19:03:25         33         19         65
28-JUL-2005 19:03:26         19         19         65
```

无论 GRP 的实际值是多少，关键在于我们要把数据按照时间分组，每 5 秒一组。然后调用聚合函数，就像最初的解决方案那样。

```
select hr,grp,sum(trx_cnt) total
  from (
select trx_date,trx_cnt,
       to_number(to_char(trx_date,'hh24')) hr,
       ceil(to_number(to_char(trx_date-1/24/60/60,'miss'))/5.0) grp
  from trx_log
       )x
 group by hr,grp

HR        GRP      TOTAL
-- ---------- ----------
19         62        141
19         63        178
19         64        202
19         65        173
```

把 TRX_DATE 对应的“小时”也作为分组列是有特殊用意的，因为交易日志可能会分布在相

邻的几个小时里。对于 DB2 和 Oracle，我们还可以调用窗口函数 SUM OVER 得出同样的结果集。下面的查询打印出了 TRX_LOG 表的全部数据，并基于“逻辑分组”生成了 TRX_CNT 的累计合计值，同时还为“逻辑分组”的每一行都计算出了当前分组的 TRX_CNT 合计值 TOTAL。

```
select trx_id, trx_date, trx_cnt,
       sum(trx_cnt)over(partition by ceil(trx_id/5.0)
                        order by trx_date
                        range between unbounded preceding
                          and current row) runing_total,
       sum(trx_cnt)over(partition by ceil(trx_id/5.0)) total,
       case when mod(trx_id,5.0) = 0 then 'X' end grp_end
  from trx_log

TRX_ID TRX_DATE                TRX_CNT RUNING_TOTAL      TOTAL GRP_END
------ -------------------- ---------- ------------ ---------- -------
     1 28-JUL-2005 19:03:07         44           44        141
     2 28-JUL-2005 19:03:08         18           62        141
     3 28-JUL-2005 19:03:09         23           85        141
     4 28-JUL-2005 19:03:10         29          114        141
     5 28-JUL-2005 19:03:11         27          141        141 X
     6 28-JUL-2005 19:03:12         45           45        178
     7 28-JUL-2005 19:03:13         45           90        178
     8 28-JUL-2005 19:03:14         32          122        178
     9 28-JUL-2005 19:03:15         41          163        178
    10 28-JUL-2005 19:03:16         15          178        178 X
    11 28-JUL-2005 19:03:17         24           24        202
    12 28-JUL-2005 19:03:18         47           71        202
    13 28-JUL-2005 19:03:19         37          108        202
    14 28-JUL-2005 19:03:20         48          156        202
    15 28-JUL-2005 19:03:21         46          202        202 X
    16 28-JUL-2005 19:03:22         44           44        173
    17 28-JUL-2005 19:03:23         36           80        173
    18 28-JUL-2005 19:03:24         41          121        173
    19 28-JUL-2005 19:03:25         33          154        173
    20 28-JUL-2005 19:03:26         19          173        173 X
```

12.18　多维度聚合运算

1. 问题

你想同时进行不同维度的聚合运算。例如，你希望得到一个结果集，其中包括每个员工的名字、部门、他所在部门的员工总数（包括他自己）、和他做相同工作的员工总数（该合计值中也包括他自己），以及 EMP 表中的员工总人数。最终的结果集如下所示。

```
ENAME  DEPTNO DEPTNO_CNT JOB        JOB_CNT  TOTAL
------ ------ ---------- --------- -------- ------
MILLER     10          3 CLERK            4     14
CLARK      10          3 MANAGER          3     14
KING       10          3 PRESIDENT        1     14
SCOTT      20          5 ANALYST          2     14
FORD       20          5 ANALYST          2     14
SMITH      20          5 CLERK            4     14
JONES      20          5 MANAGER          3     14
```

```
ADAMS      20          5 CLERK            4     14
JAMES      30          6 CLERK            4     14
MARTIN     30          6 SALESMAN         4     14
TURNER     30          6 SALESMAN         4     14
WARD       30          6 SALESMAN         4     14
ALLEN      30          6 SALESMAN         4     14
BLAKE      30          6 MANAGER          3     14
```

2. 解决方案

有了窗口函数，很容易解决本问题。对于不支持窗口函数的数据库，我们可以使用标量子查询。

DB2、Oracle 和 SQL Server

使用窗口函数 COUNT OVER，按照不同的分区或分组执行聚合运算。

```
select ename,
       deptno,
       count(*)over(partition by deptno) deptno_cnt,
       job,
       count(*)over(partition by job) job_cnt,
       count(*)over() total
  from emp
```

PostgreSQL 和 MySQL

在 SELECT 列表里使用标量子查询，基于不同的分组执行聚合运算。

```
1  select e.ename,
2         e.deptno,
3         (select count(*) from emp d
4           where d.deptno = e.deptno) as deptno_cnt,
5         job,
6         (select count(*) from emp d
7           where d.job = e.job) as job_cnt,
8         (select count(*) from emp) as total
9    from emp e
```

3. 讨论

DB2、Oracle 和 SQL Server

这个实例充分展示了窗口函数的威力和方便性。只要简单地指定好不同的聚合运算维度，就能轻松创建出非常详尽的报表，不需要一次又一次的自连接操作，也不需要在 SELECT 列表中编写冗长笨重、性能低下的子查询。窗口函数 COUNT OVER 就这样独力完成了全部工作。为了更深入地理解输出结果，你先仔细观察每个 COUNT 操作的 OVER 子句。

```
count(*)over(partition by deptno)

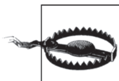
count(*)over(partition by job)

count(*)over( )
```

请记住 OVER 子句的主要组成部分：首先是分区，由 PARTITION BY 指定；然后是帧（frame）或者窗口（window），由 ORDER BY 指定。我们来看第一个 COUNT，它的分区是 DEPTNO。EMP

表的数据将按照 DEPTNO 分组，并将针对每个组执行 COUNT 操作。由于没有指定帧或窗口子句（没有 ORDER BY），分组包含的全部行都会被纳入计数范围。PARTITION BY 子句会找出所有可能的 DEPTNO 值，然后调用 COUNT 函数统计每一种 DEPTNO 对应的行数。对于本例而言，COUNT(*)OVER(PARTITION BY DEPTNO) 中的 PARTITION BY 子句能够识别出的分区值会是 10、20 和 30。

第二个 COUNT 的处理过程也是如此，只不过这次要按照 JOB 分区。最后一个 COUNT 并没有指定任何分区，只放了一个空括号。空括号是指"整张表"。因此，虽然前两个 COUNT 基于指定的分组或分区进行聚合运算，最后一个 COUNT 则会计算出整个 EMP 表的行数。



请记住 WHERE 子句执行过后窗口函数才会被评估执行。如果我们使用 WHRE 条件先行过滤掉了部分数据，例如删除了 DEPTNO 等于 10 的员工，那么 TOTAL 值将变成 11，而不再是 14。窗口函数执行过后若还想再做一些结果过滤，就必须把窗口函数查询放进内嵌视图，并从该视图中删除我们不希望看到的数据。

PostgreSQL 和 MySQL

对于主查询返回的每一行数据（EMP E 中的行），在 SELECT 列表里使用多个标量子查询基于每个 DEPTNO 和 JOB 生成不同的计数值。为得到 TOTAL 值，只要再写一个标量子查询获取 EMP 表的员工总人数即可。

12.19 动态区间聚合运算

1. 问题

你想执行动态的聚合运算，例如计算 EMP 表工资的动态合计。你希望把入职最早的员工的 HIREDATE 作为起始点，每隔 90 天计算一次工资合计值。你想调查一下在最早入职的员工和最近入职的员工之间每隔 90 天工资的波动状况。你希望得到如下所示的结果集。

```
HIREDATE        SAL SPENDING_PATTERN
----------- ------- ----------------
17-DEC-1980     800              800
20-FEB-1981    1600             2400
22-FEB-1981    1250             3650
02-APR-1981    2975             5825
01-MAY-1981    2850             8675
09-JUN-1981    2450             8275
08-SEP-1981    1500             1500
28-SEP-1981    1250             2750
17-NOV-1981    5000             7750
03-DEC-1981     950            11700
03-DEC-1981    3000            11700
23-JAN-1982    1300            10250
09-DEC-1982    3000             3000
12-JAN-1983    1100             4100
```

2. 解决方案

部分数据库支持在窗口函数的帧或窗口子句中指定动态窗口，这样一来本问题的解决就变得容易多了。关键之处在于调用窗口函数时要按照 HIREDATE 排序，并指定一个为期 90 天的

日期窗口，该日期窗口的起始点是第一个员工的入职日期。工资合计值的计算针对的是一组动态变化的员工，包括当前员工，以及入职时间早于当前员工但相差不超过 90 天的所有人。对于不支持这一类窗口函数的数据库，可以使用标量子查询，做法自然稍显繁琐。

DB2 和 Oracle

对于 DB2 和 Oracle，使用窗口函数 SUM OVER，并按照 HIREDATE 排序。在窗口或"帧"子句里指定时间范围为 90 天，这样就能针对每个员工以及入职时间比他早 90 天以内的所有人的工资执行合计计算。因为 DB2 的窗口函数不支持在 ORDER BY 子句中指定 HIREDATE（下面第 3 行代码)，我们只好改用 ORDERY BY DAYS(HIREDATE)。

```
1  select hiredat,
2         sal,
3         sum(sal)over(order by days(hiredate)
4                         range between 90 preceding
5                           and current row) spending_pattern
6    from emp e
```

相较于 DB2 解决方案，Oracle 解决方案更加简洁明了，因为 Oracle 的窗口函数支持 DATE 类型排序。

```
1  select hiredate,
2         sal,
3         sum(sal)over(order by hiredate
4                         range between 90 preceding
5                           and current row) spending_pattern
6    from emp e
```

MySQL、PostgreSQL 和 SQL Server

使用标量子查询计算工资合计值，合计值的计算范围包括当前员工，以及入职时间早于当前员工但相差不超过 90 天的所有人。

```
1 select e.hiredate,
2        e.sal,
3        (select sum(sal) from emp d
4          where d.hiredate between e.hiredate-90
5                                and e.hiredate) as spending_pattern
6   from emp e
7  order by 1
```

3. 讨论

DB2 和 Oracle

DB2 和 Oracle 的解决方案大致相同。两者差别极小，唯一的不同之处在于窗口函数的 ORDER BY 子句如何操作 HIREDATE。在写作本书时，如果想通过一个数值来指定窗口的范围，DB2 是不支持在 ORDER BY 子句中使用 DATE 类型的。（例如，RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW 支持日期类型的排序，但 RANGE BETWEEN 90 PRECEDING AND CURRENT ROW 却不支持这么做。）

掌握本解决方案的关键是要了解查询语句中窗口子句的工作原理。首先，我们定义好的窗口会按照 HIREDATE 对所有员工的工资进行排序。然后，调用聚合函数计算工资总额。但是，这里计算出的并不是所有工资的总和。具体的处理过程如下所示。

(1) 评估最早入职的员工的工资。第一个员工入职之前不会有其他人已经入职，因而此时的工资总额就是第一个员工的工资。
(2)（按照 HIREDATE 的先后顺序）评估下一位员工的工资。该员工的工资会被计入动态工资合计值，其他入职时间比他早、但相差 90 天以内的员工的工资也会被计入合计值。

第一个员工的 HIREDATE 是 1980 年 12 月 17 日，紧接着入职的下一位员工的 HIREDATE 是 1981 年 2 月 20 日。第二个员工的入职日期和第一个员工相差不足 90 天，因此第二个员工对应的动态工资合计值等于 2400（1600 + 800）。为了更深入地理解 SPENDING_PATTERN 列的计算方法，不妨仔细观察下面的查询语句及其结果集。

```
select distinct
       dense_rank()over(order by e.hiredate) window,
       e.hiredate current_hiredate,
       d.hiredate hiredate_within_90_days,
       d.sal sals_used_for_sum
  from emp e,
       emp d
 where d.hiredate between e.hiredate-90 and e.hiredate

WINDOW CURRENT_HIREDATE HIREDATE_WITHIN_90_DAYS SALS_USED_FOR_SUM
------ ---------------- ----------------------- -----------------
     1 17-DEC-1980      17-DEC-1980                           800
     2 20-FEB-1981      17-DEC-1980                           800
     2 20-FEB-1981      20-FEB-1981                          1600
     3 22-FEB-1981      17-DEC-1980                           800
     3 22-FEB-1981      20-FEB-1981                          1600
     3 22-FEB-1981      22-FEB-1981                          1250
     4 02-APR-1981      20-FEB-1981                          1600
     4 02-APR-1981      22-FEB-1981                          1250
     4 02-APR-1981      02-APR-1981                          2975
     5 01-MAY-1981      20-FEB-1981                          1600
     5 01-MAY-1981      22-FEB-1981                          1250
     5 01-MAY-1981      02-APR-1981                          2975
     5 01-MAY-1981      01-MAY-1981                          2850
     6 09-JUN-1981      02-APR-1981                          2975
     6 09-JUN-1981      01-MAY-1981                          2850
     6 09-JUN-1981      09-JUN-1981                          2450
     7 08-SEP-1981      08-SEP-1981                          1500
     8 28-SEP-1981      08-SEP-1981                          1500
     8 28-SEP-1981      28-SEP-1981                          1250
     9 17-NOV-1981      08-SEP-1981                          1500
     9 17-NOV-1981      28-SEP-1981                          1250
     9 17-NOV-1981      17-NOV-1981                          5000
    10 03-DEC-1981      08-SEP-1981                          1500
    10 03-DEC-1981      28-SEP-1981                          1250
    10 03-DEC-1981      17-NOV-1981                          5000
    10 03-DEC-1981      03-DEC-1981                           950
    10 03-DEC-1981      03-DEC-1981                          3000
    11 23-JAN-1982      17-NOV-1981                          5000
    11 23-JAN-1982      03-DEC-1981                           950
    11 23-JAN-1982      03-DEC-1981                          3000
    11 23-JAN-1982      23-JAN-1982                          1300
    12 09-DEC-1982      09-DEC-1982                          3000
    13 12-JAN-1983      09-DEC-1982                          3000
    13 12-JAN-1983      12-JAN-1983                          1100
```

仔细看 WINDOW 列的话，我们会发现具有相同 WINDOW 值的行会被计入动态工资合计值。以 WINDOW 值等于 3 的那一组为例，该窗口中被计入合计的工资分别是 800、1600 和 1250，合计为 3650。回头再看一下“问题”部分给出的最终结果集，我们看到 1981 年 2 月 22 日 (WINDOW 3) 那一行的 SPENDING_PATTERN 值确实是 3650。如果想证明自连接查询为每个窗口都找到了正确的工资，我们只要按照 CURRENT_DATE 分组，并把 SALS_USED_FOR_SUM 值相加求和即可。查询结果应该会和“问题”部分给出的结果集相一致（1981 年 12 月 3 日对应的结果本应有两行，但这里把重复行去掉了）。

```
select current_hiredate,
       sum(sals_used_for_sum) spending_pattern
  from (
select distinct
       dense_rank()over(order by e.hiredate) window,
       e.hiredate current_hiredate,
       d.hiredate hiredate_within_90_days,
       d.sal sals_used_for_sum
  from emp e,
       emp d
 where d.hiredate between e.hiredate-90 and e.hiredate
       ) x
 group by current_hiredate

CURRENT_HIREDATE SPENDING_PATTERN
---------------- ----------------
17-DEC-1980                   800
20-FEB-1981                  2400
22-FEB-1981                  3650
02-APR-1981                  5825
01-MAY-1981                  8675
09-JUN-1981                  8275
08-SEP-1981                  1500
28-SEP-1981                  2750
17-NOV-1981                  7750
03-DEC-1981                 11700
23-JAN-1982                 10250
09-DEC-1982                  3000
12-JAN-1983                  4100
```

MySQL、PostgreSQL 和 SQL Server

基于 HIREDATE 排序，并使用聚合函数 SUM 以计算出每 90 天的工资合计值。除此之外，本解决方案的关键之处还包括标量子查询的运用（自连接查询也能达到同样目的）。为了方便你理解，不妨把本解决方案先转换成自连接查询，并仔细观察哪些行会被包含进求和计算。考虑如下所示的结果集，它的查询结果和前面给出的解决方案相同。

```
select e.hiredate,
       e.sal,
       sum(d.sal) as spending_pattern
  from emp e, emp d
 where d.hiredate
       between e.hiredate-90 and e.hiredate
 group by e.hiredate,e.sal
 order by 1
```

```
HIREDATE      SAL SPENDING_PATTERN
----------- ----- ----------------
17-DEC-1980   800              800
20-FEB-1981  1600             2400
22-FEB-1981  1250             3650
02-APR-1981  2975             5825
01-MAY-1981  2850             8675
09-JUN-1981  2450             8275
08-SEP-1981  1500             1500
28-SEP-1981  1250             2750
17-NOV-1981  5000             7750
03-DEC-1981   950            11700
03-DEC-1981  3000            11700
23-JAN-1982  1300            10250
09-DEC-1982  3000             3000
12-JAN-1983  1100             4100
```

如果上述输出结果依然不够清楚明白，不妨删除分组和聚合运算部分，先打印出笛卡儿积的结果。首先使用 EMP 表产生一个笛卡儿积以便每个 HIREDATE 都可以和其他 HIREDATE 进行比较。（下面只打印出了部分结果集，因为 EMP 表的笛卡儿积会返回 196 行（14 × 14）数据。）

```
select e.hiredate,
       e.sal,
       d.sal,
       d.hiredate
  from emp e, emp d

HIREDATE      SAL   SAL HIREDATE
----------- ----- ----- -----------
17-DEC-1980   800   800 17-DEC-1980
17-DEC-1980   800  1600 20-FEB-1981
17-DEC-1980   800  1250 22-FEB-1981
17-DEC-1980   800  2975 02-APR-1981
17-DEC-1980   800  1250 28-SEP-1981
17-DEC-1980   800  2850 01-MAY-1981
17-DEC-1980   800  2450 09-JUN-1981
17-DEC-1980   800  3000 09-DEC-1982
17-DEC-1980   800  5000 17-NOV-1981
17-DEC-1980   800  1500 08-SEP-1981
17-DEC-1980   800  1100 12-JAN-1983
17-DEC-1980   800   950 03-DEC-1981
17-DEC-1980   800  3000 03-DEC-1981
17-DEC-1980   800  1300 23-JAN-1982
20-FEB-1981  1600   800 17-DEC-1980
20-FEB-1981  1600  1600 20-FEB-1981
20-FEB-1981  1600  1250 22-FEB-1981
20-FEB-1981  1600  2975 02-APR-1981
20-FEB-1981  1600  1250 28-SEP-1981
20-FEB-1981  1600  2850 01-MAY-1981
20-FEB-1981  1600  2450 09-JUN-1981
20-FEB-1981  1600  3000 09-DEC-1982
20-FEB-1981  1600  5000 17-NOV-1981
```

```
20-FEB-1981  1600  1500 08-SEP-1981
20-FEB-1981  1600  1100 12-JAN-1983
20-FEB-1981  1600   950 03-DEC-1981
20-FEB-1981  1600  3000 03-DEC-1981
20-FEB-1981  1600  1300 23-JAN-1982
```

仔细查看上述结果集可以发现，除了其自身，不存在比 1980 年 12 月 17 日更早的 HIREDATE。因此，那一行对应的合计值应该是 800。然后是下一个 HIREDATE，1981 年 2 月 20 日，我们注意到只有一个 HIREDATE 落在了 90 天的窗口范围内（比该日期早 90 天），那就是 1980 年 12 月 17 日。为 1980 年 12 月 17 日对应的 SAL 值和 1981 年 2 月 20 对应的 SAL 值求和，结果是 2400（因为我们寻找的是当前 HIREDATE 以及 90 天之内的日期），这恰好就是最终结果集该 HIREDATE 对应的工资合计值。

现在我们已经理解具体的做法了，接着在 WHERE 子句里添加一个条件以筛选出每一个 HIREDATE 以及早于该 HIREDATE 90 天以内的日期。

```
select e.hiredate,
       e.sal,
       d.sal sal_to_sum,
       d.hiredate within_90_days
  from emp e, emp d
 where d.hiredate
       between e.hiredate-90 and e.hiredate
 order by 1

HIREDATE      SAL SAL_TO_SUM WITHIN_90_DAYS
----------- ----- ---------- --------------
17-DEC-1980   800        800    17-DEC-1980
20-FEB-1981  1600        800    17-DEC-1980
20-FEB-1981  1600       1600    20-FEB-1981
22-FEB-1981  1250        800    17-DEC-1980
22-FEB-1981  1250       1600    20-FEB-1981
22-FEB-1981  1250       1250    22-FEB-1981
02-APR-1981  2975       1600    20-FEB-1981
02-APR-1981  2975       1250    22-FEB-1981
02-APR-1981  2975       2975    02-APR-1981
01-MAY-1981  2850       1600    20-FEB-1981
01-MAY-1981  2850       1250    22-FEB-1981
01-MAY-1981  2850       2975    02-APR-1981
01-MAY-1981  2850       2850    01-MAY-1981
09-JUN-1981  2450       2975    02-APR-1981
09-JUN-1981  2450       2850    01-MAY-1981
09-JUN-1981  2450       2450    09-JUN-1981
08-SEP-1981  1500       1500    08-SEP-1981
28-SEP-1981  1250       1500    08-SEP-1981
28-SEP-1981  1250       1250    28-SEP-1981
17-NOV-1981  5000       1500    08-SEP-1981
17-NOV-1981  5000       1250    28-SEP-1981
17-NOV-1981  5000       5000    17-NOV-1981
03-DEC-1981   950       1500    08-SEP-1981
03-DEC-1981   950       1250    28-SEP-1981
03-DEC-1981   950       5000    17-NOV-1981
03-DEC-1981   950        950    03-DEC-1981
```

```
03-DEC-1981   950       3000    03-DEC-1981
03-DEC-1981  3000       1500    08-SEP-1981
03-DEC-1981  3000       1250    28-SEP-1981
03-DEC-1981  3000       5000    17-NOV-1981
03-DEC-1981  3000        950    03-DEC-1981
03-DEC-1981  3000       3000    03-DEC-1981
23-JAN-1982  1300       5000    17-NOV-1981
23-JAN-1982  1300        950    03-DEC-1981
23-JAN-1982  1300       3000    03-DEC-1981
23-JAN-1982  1300       1300    23-JAN-1982
09-DEC-1982  3000       3000    09-DEC-1982
12-JAN-1983  1100       3000    09-DEC-1982
12-JAN-1983  1100       1100    12-JAN-1983
```

现在我们能清楚地看到哪些 SAL 值会被计入合计值了，接下来只要调用聚合函数 SUM 生成最终结果集即可。

```
select e.hiredate,
       e.sal,
       sum(d.sal) as spending_pattern
  from emp e, emp d
 where d.hiredate
       between e.hiredate-90 and e.hiredate
 group by e.hiredate,e.sal
 order by 1
```

比较上述查询和下面的查询（本例给出的标量子查询解决方案）的结果集，我们会发现它们其实是相同的。

```
select e.hiredate,
       e.sal,
       (select sum(sal) from emp d
         where d.hiredate between e.hiredate-90
                              and e.hiredate) as spending_pattern
  from emp e
 order by 1

HIREDATE      SAL SPENDING_PATTERN
----------- ----- ----------------
17-DEC-1980   800              800
20-FEB-1981  1600             2400
22-FEB-1981  1250             3650
02-APR-1981  2975             5825
01-MAY-1981  2850             8675
09-JUN-1981  2450             8275
08-SEP-1981  1500             1500
28-SEP-1981  1250             2750
17-NOV-1981  5000             7750
03-DEC-1981   950            11700
03-DEC-1981  3000            11700
23-JAN-1982  1300            10250
09-DEC-1982  3000             3000
12-JAN-1983  1100             4100
```

12.20 变换带有小计的结果集

1. 问题

你想创建一个包含小计的报表，并希望把该结果重新格式化，使之更具可读性。例如，你已经按照要求创建了一张报表，其中包含每个部门的所有管理者，以及每个管理者下属所有员工的工资合计值。同时，你也希望能看到两种额外的小计：去掉管理者之后、每个部门的所有下属员工的工资总额，以及上述结果集中所有工资的合计值（在部门工资小计的基础上再求和，得出总计）。现在，你得到的报表如下所示。

```
DEPTNO        MGR        SAL
------ ---------- ----------
    10       7782       1300
    10       7839       2450
    10                  3750
    20       7566       6000
    20       7788       1100
    20       7839       2975
    20       7902        800
    20                 10875
    30       7698       6550
    30       7839       2850
    30                  9400
                       24025
```

你想提供一个更具可读性的报表，因此希望把上述结果转换成如下所示的形式，以便更清楚明白地表达每个小计的含义。

```
MGR      DEPT10     DEPT20     DEPT30      TOTAL
---- ---------- ---------- ---------- ----------
7566          0       6000          0
7698          0          0       6550
7782       1300          0          0
7788          0       1100          0
7839       2450       2975       2850
7902          0        800          0
           3750      10875       9400      24025
```

2. 解决方案

首先使用 GROUP BY 的 ROLLUP 扩展生成小计，接着执行一次结果集变换（使用聚合运算和 CASE 表达式）以创建出新报表所需的列。GROUPING 函数能帮助我们方便地识别出哪些是小计（也就是说，小计都是由 ROLLUP 产生出来的，正常的 GROUP BY 无法生成小计）。由于不同的数据库对 Null 排序的处理方式各有不同，我们可能需要为某些解决方案增加 ORDER BY，以便最终的结果集和上述目标结果集相一致。

DB2 和 Oracle

使用 GROUP BY 的 ROLLUP 扩展，然后借助 CASE 表达式把数据格式化成更具可读性的报表。

```
1  select mgr,
2         sum(case deptno when 10 then sal else 0 end) dept10,
3         sum(case deptno when 20 then sal else 0 end) dept20,
```

```
 4         sum(case deptno when 30 then sal else 0 end) dept30,
 5         sum(case flag   when '11' then sal else null end) total
 6   from (
 7  select deptno,mgr,sum(sal) sal,
 8         cast(grouping(deptno) as char(1))||
 9         cast(grouping(mgr)    as char(1)) flag
10    from emp
11   where mgr is not null
12   group by rollup(deptno,mgr)
13         ) x
14   group by mgr
```

SQL Server

使用 GROUP BY 的 ROLLUP 扩展，然后借助 CASE 表达式把数据格式化成更具可读性的报表。

```
 1  select mgr,
 2         sum(case deptno when 10 then sal else 0 end) dept10,
 3         sum(case deptno when 20 then sal else 0 end) dept20,
 4         sum(case deptno when 30 then sal else 0 end) dept30,
 5         sum(case flag   when '11' then sal else null end) total
 6   from (
 7  select deptno,mgr,sum(sal) sal,
 8         cast(grouping(deptno) as char(1))+
 9         cast(grouping(mgr)    as char(1)) flag
10    from emp
11   where mgr is not null
12   group by deptno,mgr with rollup
13         ) x
14   group by mgr
```

MySQL 和 PostgreSQL

这两种数据库都不支持 GROUPING 函数。

3. 讨论

以上两种解决方案大致相同，但在字符串连接和 GROUPING 函数调用的方式上略有不同。由于它们非常相似，下面的讨论部分在展示中间结果的时候将以 SQL Server 解决方案为准（也会兼顾 DB2 和 Oracle）。

首先为每个 DEPTNO 的每个 MGR 计算出其下属员工的 SAL 合计值。基本的想法是希望打印出某个部门特定管理者下属所有员工的工资合计值。例如，下面的查询可以比较 KING 下属的 DEPTNO 10 和 DEPTNO 30 的所有员工的工资。

```
select deptno,mgr,sum(sal) sal
  from emp
 where mgr is not null
 group by mgr,deptno
 order by 1,2

DEPTNO        MGR        SAL
------ ---------- ----------
    10       7782       1300
    10       7839       2450
    20       7566       6000
```

```
    20       7788       1100
    20       7839       2975
    20       7902        800
    30       7698       6550
    30       7839       2850
```

然后使用 GROUP BY 的 ROLLUP 扩展，计算每个 DEPTNO 对应的工资小计以及全体员工（不包括最高管理者）的工资总额。

```
select deptno,mgr,sum(sal) sal
  from emp
 where mgr is not null
 group by deptno,mgr with rollup

DEPTNO        MGR        SAL
------ ---------- ----------
    10       7782       1300
    10       7839       2450
    10                  3750
    20       7566       6000
    20       7788       1100
    20       7839       2975
    20       7902        800
    20                 10875
    30       7698       6550
    30       7839       2850
    30                  9400
                       24025
```

计算出了小计之后，我们需要知道如何判断哪些值是（由 ROLLUP 产生的）小计，哪些值是由正常的 GROUP BY 产生的结果。使用 GROUPING 函数创建位图可以把小计从正常的聚合运算值中分离出来。

```
select deptno,mgr,sum(sal) sal,
       cast(grouping(deptno) as char(1))+
       cast(grouping(mgr)    as char(1)) flag
  from emp
 where mgr is not null
 group by deptno,mgr with rollup

DEPTNO        MGR        SAL FLAG
------ ---------- ---------- ----
    10       7782       1300 00
    10       7839       2450 00
    10                  3750 01
    20       7566       6000 00
    20       7788       1100 00
    20       7839       2975 00
    20       7902        800 00
    20                 10875 01
    30       7698       6550 00
    30       7839       2850 00
    30                  9400 01
                       24025 11
```

如果要讲得更清楚明白一点的话，就是 FLAG 值等于 00 的行都是正常的聚合运算结果。FLAG 值等于 01 的行是由 ROLLUP 生成的、每个 DEPTNO 的 SAL 合计值（因为 DEPTNO 是 ROLLUP 的第一个参数；如果变换一下顺序，例如改为“GROUP BY MGR, DEPTNO WITH ROLLUP”，结果会大相径庭）。FLAG 值等于 11 的行是由 ROLLUP 产生的、所有 SAL 的合计值。

现在，所需的数据都已经准备好了，接下来要使用 CASE 表达式把它们重新格式化为一个更直观的结果集。我们的目标是为跨部门的每位管理者显示其下属员工的工资合计值。如果一位管理者在某个部门没有下属员工，那就返回 0；否则，将返回该管理者在给定部门所有下属员工的工资合计值。除此之外，我们还要在报表的最后面增加一个 TOTAL 列以打印报表中全部工资的总计。满足上述所有要求的解决方案如下所示。

```
select mgr,
       sum(case deptno when 10 then sal else 0 end) dept10,
       sum(case deptno when 20 then sal else 0 end) dept20,
       sum(case deptno when 30 then sal else 0 end) dept30,
       sum(case flag   when '11' then sal else null end) total
  from (
select deptno,mgr,sum(sal) sal,
       cast(grouping(deptno) as char(1))+
       cast(grouping(mgr)    as char(1)) flag
  from emp
 where mgr is not null
 group by deptno,mgr with rollup
       ) x
 group by mgr
 order by coalesce(mgr,9999)

MGR      DEPT10     DEPT20     DEPT30      TOTAL
---- ---------- ---------- ---------- ----------
7566          0       6000          0
7698          0          0       6550
7782       1300          0          0
7788          0       1100          0
7839       2450       2975       2850
7902          0        800          0
           3750      10875       9400      24025
```

第 1 3 章

层次查询

本章介绍的实例展现了数据中可能存在的层次关系。通常而言，以层次关系的形式访问并展示数据相较于存储这些数据要困难得多。而且，由于SQL缺乏足够的弹性（SQL的非递归性质），更使得这种状况雪上加霜。当涉及层次查询时，充分利用关系数据库管理系统提供的相关特性毫无疑问是非常重要的；否则在耗费了大量时间和精力构造出复杂、费解的数据模型，并编写出了一些低效的查询语句之后，我们可能会最终发现这些都无济于事。对于PostgreSQL用户， WITH递归查询很可能会被加入到后续的版本里，因此不妨关注一下本章给出的DB2解决方案。

本章提供的实例将充分利用各种关系数据库管理系统提供的函数把具有层次结构的数据分解开。在开始讲述具体的实例之前，我们先来看一下 EMP 表以及 EMPNO 和 MGR 之间的层次关系。

```
select empno,mgr
  from emp
order by 2

     EMPNO        MGR
---------- ----------
      7788       7566
      7902       7566
      7499       7698
      7521       7698
      7900       7698
      7844       7698
      7654       7698
      7934       7782
      7876       7788
      7566       7839
      7782       7839
```

```
7698        7839
7369        7902
7839
```

如果仔细观察的话，我们会发现 MGR 值其实也是一个 EMPNO。也就是说，每个员工的管理者也会作为一个员工存在于 EMP 表里，他们并没有被存储到其他地方。MGR 和 EMPNO 之间的关系类似于父子关系，对于每一个给定的 EMPNO，它对应的 MGR 值就是他的直接父节点。（一个员工的管理者也可能会有更高级的管理者，以此类推，这就产生一个多层的结构。）对于没有管理者的员工，他的 MGR 值就是 Null。

13.1 展现父子关系

1. 问题

你想找出子节点对应的父节点信息。例如，你希望显示每个员工及其管理者的名字。你希望返回的结果集如下所示。

```
EMPS_AND_MGRS
------------------------------
FORD works for JONES
SCOTT works for JONES
JAMES works for BLAKE
TURNER works for BLAKE
MARTIN works for BLAKE
WARD works for BLAKE
ALLEN works for BLAKE
MILLER works for CLARK
ADAMS works for SCOTT
CLARK works for KING
BLAKE works for KING
JONES works for KING
SMITH works for FORD
```

2. 解决方案

基于 MGR 和 EMPNO 对 EMP 表做自连接查询，找出每个员工的管理者的名字。然后使用数据库提供的字符串连接函数生成符合要求的字符串。

DB2、Oracle 和 PostgreSQL

自连接 EMP 表。然后使用双竖线 || 连接字符串。

```
1 select a.ename || ' works for ' || b.ename as emps_and_mgrs
2   from emp a, emp b
3  where a.mgr = b.empno
```

MySQL

自连接 EMP 表，然后使用 CONCAT 函数连接字符串。

```
1 select concat(a.ename, ' works for ',b.ename) as emps_and_mgrs
2   from emp a, emp b
3  where a.mgr = b.empno
```

SQL Server

自连接 EMP 表，然后使用加号“+”连接字符串。

```
1 select a.ename + ' works for ' + b.ename as emps_and_mgrs
2   from emp a, emp b
3  where a.mgr = b.empno
```

3. 讨论

以上所有解决方案的实现思路大致相同。不同之处仅在于字符串连接的方式，所以下面的讨论能够兼顾到所有的解决方案，不用再一一解释每一个具体的做法。

基于 MGR 和 EMPNO 的连接查询是关键所在。首先通过自连接 EMP 表创建一个笛卡儿积（下面只显示了笛卡儿积返回值的一部分）。

```
select a.empno, b.empno
  from emp a, emp b

EMPNO        MGR
----- ----------
 7369       7369
 7369       7499
 7369       7521
 7369       7566
 7369       7654
 7369       7698
 7369       7782
 7369       7788
 7369       7839
 7369       7844
 7369       7876
 7369       7900
 7369       7902
 7369       7934
 7499       7369
 7499       7499
 7499       7521
 7499       7566
 7499       7654
 7499       7698
 7499       7782
 7499       7788
 7499       7839
 7499       7844
 7499       7876
 7499       7900
 7499       7902
 7499       7934
```

如上所示，笛卡儿积返回了每一种可能的 EMPNO/EMPNO 组合。（这样看起来所有人都是 EMPNO 7369 的管理者，甚至也包括 7369 本人。）

下一步要过滤结果集，以便只返回每个员工及其管理者的 EMPNO。增加 MGR 和 EMPNO 相等的判断条件即可。

```
1  select a.empno, b.empno mgr
2    from emp a, emp b
3   where a.mgr = b.empno

     EMPNO        MGR
---------- ----------
      7902       7566
      7788       7566
      7900       7698
      7844       7698
      7654       7698
      7521       7698
      7499       7698
      7934       7782
      7876       7788
      7782       7839
      7698       7839
      7566       7839
      7369       7902
```

现在已经得到了每个员工及其管理者的 EMPNO，接下来只要查询 B.ENAME 提取出每个管理者的名字即可。如果你还没有完全理解本解决方案的原理，那么可以先忽略自连接方案，转而尝试下面的标量子查询方案。

```
select a.ename,
       (select b.ename
          from emp b
         where b.empno = a.mgr) as mgr
  from emp a

ENAME      MGR
---------- ----------
SMITH      FORD
ALLEN      BLAKE
WARD       BLAKE
JONES      KING
MARTIN     BLAKE
BLAKE      KING
CLARK      KING
SCOTT      JONES
KING
TURNER     BLAKE
ADAMS      SCOTT
JAMES      BLAKE
FORD       JONES
MILLER     CLARK
```

上述标量子查询方案实际上等价于自连接方案，只有一行数据除外：员工 KING 出现在了结果集里，而自连接方案的查询结果里却不包括该员工。你可能会问："为什么不会包括？"注意，Null 不等于任何值，甚至不等于它自身。在自连接解决方案中，我们基于 EMPNO 和 MGR 做相等连接查询（equi-join），这就剔除了 MGR 值等于 Null 的员工。如果既使用自连接方案，又希望员工 KING 出现在最终的结果集里，就必须使用外连接，就像下面的两种查询那样。第一种解决方案使用了 ANSI 外连接，第二种则用了 Oracle 的外连接语法。这两种解决方案的结果自然是相同的，我们把输出结果放在了第二个方案的后面。

```
/* ANSI */
select a.ename, b.ename mgr
  from emp a left join emp b
    on (a.mgr = b.empno)

/* Oracle */
select a.ename, b.ename mgr
  from emp a, emp b
 where a.mgr = b.empno (+)

ENAME      MGR
---------- ----------
FORD       JONES
SCOTT      JONES
JAMES      BLAKE
TURNER     BLAKE
MARTIN     BLAKE
WARD       BLAKE
ALLEN      BLAKE
MILLER     CLARK
ADAMS      SCOTT
CLARK      KING
BLAKE      KING
JONES      KING
SMITH      FORD
KING
```

13.2 展现祖孙关系

1. 问题

员工 CLARK 是 KING 的下属，你已经在前一个实例里学到了如何展现这一关系。那么，如果员工 CLARK 同时也是另一个员工的管理者的话，又该如何呢？考虑如下所示的查询。

```
select ename,empno,mgr
  from emp
 where ename in ('KING','CLARK','MILLER')

ENAME           EMPNO        MGR
---------- ---------- ----------
CLARK            7782       7839
KING             7839
MILLER           7934       7782
```

如上所示，员工 MILLER 是 CLARK 的下属，而 CLARK 又是 KING 的下属。你希望展现从 MILLER 到 KING 整个层次关系。你希望返回如下所示的结果集。

```
LEAF___BRANCH_ _ _ROOT
---------------------
MILLER-->CLARK-->KING
```

然而，如果还像前一个实例那样只使用一个自连接查询的话，其实无法显示从上到下整个层次关系。当然你也可以使用两个自连接查询，但你真正需要的可能是能够遍历整个层次

关系的通用做法。

2. 解决方案

本实例不同于前一个实例，正如标题所示，现在我们面对的是一个 3 层的层次关系。有些数据库没有提供能够遍历树形结构数据的函数，我们可以采用 PostgreSQL 和 MySQL 解决方案的做法，但必须再增加一个自连接查询。DB2、SQL Server 和 Oracle 为展现层次关系提供了专门的函数。因此，尽管自连接方案仍然适用于这些数据库，但已经没有必要那么做了。

DB2 和 SQL Server

使用 `WITH` 递归查询找到 MILLER 的管理者 CLARK，以及 CLARK 的管理者 KING。下面的解决方案用到了 SQL Server 的字符串连接操作符 `+`。

```
1    with x (tree,mgr,depth)
2      as (
3  select cast(ename as varchar(100)),
4          mgr, 0
5    from emp
6   where ename = 'MILLER'
7   union all
8  select cast(x.tree+'-->'+e.ename as varchar(100)),
9          e.mgr, x.depth+1
10   from emp e, x
11  where x.mgr = e.empno
12 )
13 select tree leaf___branch___root
14   from x
15  where depth = 2
```

对于 DB2 数据库，上述解决方案唯一需要修改的是改用 DB2 的字符串连接操作符 `||`。除此以外，该解决方案完全适用于 DB2 和 SQL Server。

Oracle

使用 `SYS_CONNECT_BY_PATH` 函数返回 MILLER、MILLER 的管理者 CLARK 和 CLARK 的管理者 KING。使用 `CONNECT BY` 子句遍历树形结构。

```
1  select ltrim(
2           sys_connect_by_path(ename,'-->'),
3         '-->') leaf___branch___root
4    from emp
5   where level = 3
6   start with ename = 'MILLER
7 connect by prior mgr = empno
```

PostgreSQL 和 MySQL

基于 `EMP` 表自连接两次返回 MILLER、MILLER 的管理者 CLARK 以及 CLARK 的管理者 KING。下面的解决方案使用了 PostgreSQL 的双竖线 `||` 字符串连接操作符。

```
1  select a.ename||'-->'||b.ename
2                ||'-->'||c.ename as leaf___branch___root
3    from emp a, emp b, emp c
```

```
4   where a.ename = 'MILLER'
5     and a.mgr = b.empno
6     and b.mgr = c.empno
```

对于 MySQL 用户，要使用 CONCAT 函数，这样上述解决方案就可以像 PostgreSQL 一样正常运行了。

3. 讨论

DB2 和 SQL Server

本解决方案的做法是从叶子节点开始遍历到根节点（你不妨尝试从相反的方向遍历各个节点，这是一次很好的练习）。UNION ALL 的前半部分仅仅找出了员工 MILLER（叶子节点）所在的行。UNION ALL 的后半部分找到了 MILLER 的管理者，然后再找到那个人的管理者，这种找出“管理者的管理者”的处理会一直重复，直至找到最高级别的管理者（根节点）才会停下。DEPTH 值从 0 开始，每找到一层管理者就自动加 1。当执行递归查询时，DB2 会为我们维护一个 DEPTH 值。



如果对 WITH 递归查询感兴趣的话，请参考 Jonathan Gennick 写的一篇有趣且有深度的介绍性文章“Understanding the WITH Clause”，网址是 http://gennick.com/database/understanding-the-with-clause。

接下来，UNION ALL 后半部分的查询把递归视图 X 连接到 EMP 表，从而定义父子关系。下面的查询语句使用了 SQL Server 的字符串连接操作符。

```
  with x (tree,mgr,depth)
    as (
select cast(ename as varchar(100)),
       mgr, 0
  from emp
 where ename = 'MILLER'
 union all
select cast(x.tree+'-->'+e.ename as varchar(100)),
       e.mgr, x.depth+1
  from emp e, x
 where x.mgr = e.empno
)
select tree leaf___branch___root
  from x

TREE            DEPTH
---------- ----------
MILLER              0
CLARK               1
KING                2
```

现在，最重要的问题已经解决了。从 MILLER 开始，我们找出了从下到上的整个层次关系。下面只需要格式化即可。因为树形结构的遍历是递归的，只要把 EMP 表的当前 ENAME 连接到它前面的一个名字即可，如下所示。

```
  with x (tree,mgr,depth)
    as (
```

```
select cast(ename as varchar(100)),
       mgr, 0
  from emp
 where ename = 'MILLER'
 union all
select cast(x.tree+'-->'+e.ename as varchar(100)),
       e.mgr, x.depth+1
  from emp e, x
 where x.mgr = e.empno
)
select depth, tree
  from x

DEPTH TREE
----- ----------------------
    0 MILLER
    1 MILLER-->CLARK
    2 MILLER-->CLARK-->KING
```

最后，只需要保留层次结构中的最后一行。有多种方法可以实现这一点，但本解决方案通过 DEPTH 来判断何时到达了根节点。（显然，如果 CLARK 的管理者不是 KING，那么基于 DEPTH 的过滤条件可能就不适用。13.3 节提供一种更通用的方法，它不需要上述过滤条件。）

Oracle

对于 Oracle 解决方案而言，CONNECT BY 子句完成了全部的工作。从 MILLER 开始，我们不需要任何连接查询就能一直遍历到 KING。CONNECT BY 子句里的表达式定义了数据之间的关系以及如何遍历树形结构。

```
 select ename
   from emp
  start with ename = 'MILLER'
connect by prior mgr = empno

ENAME
----------
MILLER
CLARK
KING
```

关键字 PRIOR 可以访问层次关系中前一条记录的值。因此，对于任意给定的 EMPNO，可以使用 PRIOR MGR 去获得前一个员工的管理者的 EMPNO。对于本实例而言，当看到 CONNECT BY PRIOR MGR = EMPNO 这样的子句时，要把它当作是父记录和子记录之间的连接查询。



更多关于 CONNECT BY 及其相关功能的内容，请参考来自 Oracle Technology Network 的文章：“Querying Hierarchies:Top-of-the-Line Support”。

现在，已经成功地显示了从 MILLER 到 KING 的整个层次关系。大部分问题都已经解决了。下面只需格式化即可。使用 SYS_CONNECT_BY_PATH 函数把所有 ENAME 逐一地连接起来。

```
 select sys_connect_by_path(ename,'-->') tree
   from emp
  start with ename = 'MILLER'
connect by prior mgr = empno

TREE
---------------------------
-->MILLER
-->MILLER-->CLARK
-->MILLER-->CLARK-->KING
```

因为只希望保留完整的层次关系，我们可以通过伪列 LEVEL 过滤掉不需要的数据（下一个实例将给出一种更加通用的解决方案）。

```
select sys_connect_by_path(ename,'-->') tree
  from emp
 where level = 3
 start with ename = 'MILLER'
connect by prior mgr = empno

TREE
-----------------------------
-->MILLER-->CLARK-->KING
```

最后，使用函数 LTRIM 从结果集中删除最前面的 -->。

PostgreSQL 和 MySQL

因为没有层次查询相关的内置支持，所以我们必须自连接 *n* 次以返回整个树形结构。（此处的 *n* 代表叶子节点和根节点之间节点的总个数，包括根节点自身。在本实例中，对于 MILLER 而言，CLARK 是一个分支节点（branch node），而 KING 则是根节点，因此从 MILLER 到 KING 的距离是两个节点，*n*=2。）本解决方案借用了前一个实例的技巧，只是多了一个自连接。

```
select a.ename as leaf,
       b.ename as branch,
       c.ename as root
  from emp a, emp b, emp c
 where a.ename = 'MILLER'
   and a.mgr = b.empno
   and b.mgr = c.empno

LEAF       BRANCH     ROOT
---------- ---------- -----
MILLER     CLARK      KING
```

下一步也是最后一步，即格式化输出结果，PostgreSQL 使用 || 字符串连接操作符，MySQL 则使用 CONCAT 函数。这一类查询的缺点是，如果层次关系发生了变化，例如，在 CLARK 和 KING 之间多了一个节点，我们就得再增加一个自连接以返回整个树形结构。这就不如有专门的内置函数协助处理层次关系遍历的数据库方便。

13.3　创建层次视图

1. 问题

你想用一个结果集展示一个表的全部数据之间的层次关系。比如 EMP 表，员工 KING 没有管理者，因此 KING 是根节点。你希望展示从 KING 开始，所有 KING 的下属以及 KING 的下属的下属（如果有的话）。最终，你希望得到如下所示的结果集。

```
EMP_TREE
------------------------------
KING
KING - BLAKE
KING - BLAKE - ALLEN
KING - BLAKE - JAMES
KING - BLAKE - MARTIN
KING - BLAKE - TURNER
KING - BLAKE - WARD
KING - CLARK
KING - CLARK - MILLER
KING - JONES
KING - JONES - FORD
KING - JONES - FORD - SMITH
KING - JONES - SCOTT
KING - JONES - SCOTT – ADAMS
```

2. 解决方案

DB2 和 SQL Server

使用 WITH 递归查询从 KING 开始构造层次关系，进而最终展现出所有员工之间的关系。下面的解决方案使用了 DB2 的字符串连接操作符 ||。SQL Server 用户则需要使用字符串连接操作符 +。除了字符串连接操作符的不同，本解决方案同时适用于这两种数据库。

```
 1   with x (ename,empno)
 2      as (
 3  select cast(ename as varchar(100)),empno
 4    from emp
 5   where mgr is null
 6   union all
 7  select cast(x.ename||' - '||e.ename as varchar(100)),
 8         e.empno
 9    from emp e, x
10   where e.mgr = x.empno
11  )
12  select ename as emp_tree
13    from x
14   order by 1
```

Oracle

使用 CONNECT BY 函数定义层次关系，接着使用 SYS_CONNECT_BY_PATH 函数格式化输出结果。

```
1  select ltrim(
2           sys_connect_by_path(ename,' - '),
3         ' - ') emp_tree
```

```
4    from emp
5   start with mgr is null
6 connect by prior empno=mgr
7   order by 1
```

本解决方案不同于前一个实例，它不需要基于伪列 LEVEL 的过滤条件。因为没有该过滤条件，所有可能的树形结构都会被显示出来（此处 PRIOR EMPNO=MGR）。

PostgreSQL

使用 3 个 UNION 和多个自连接。

```
 1  select emp_tree
 2    from (
 3  select ename as emp_tree
 4    from emp
 5   where mgr is null
 6  union
 7  select a.ename||' - '||b.ename
 8    from emp a
 9         join
10         emp b on (a.empno=b.mgr)
11   where a.mgr is null
12 union
13 select rtrim(a.ename||' - '||b.ename
14                    ||' - '||c.ename,' - ')
15   from emp a
16        join
17        emp b on (a.empno=b.mgr)
18        left join
19        emp c on (b.empno=c.mgr)
20  where a.ename = 'KING'
21 union
22 select rtrim(a.ename||' - '||b.ename||' - '||
23              c.ename||' - '||d.ename,' - ')
24   from emp a
25        join
26        emp b on (a.empno=b.mgr)
27        join
28        emp c on (b.empno=c.mgr)
29        left join
30        emp d on (c.empno=d.mgr)
31  where a.ename = 'KING'
32        ) x
33  where tree is not null
34  order by 1
```

MySQL

使用 3 个 UNION 和多个自连接。

```
 1  select emp_tree
 2    from (
 3  select ename as emp_tree
 4    from emp
 5   where mgr is null
 6  union
```

```
 7 select concat(a.ename,' - ',b.ename)
 8   from emp a
 9        join
10        emp b on (a.empno=b.mgr)
11  where a.mgr is null
12 union
13 select concat(a.ename,' - ',
14               b.ename,' - ',c.ename)
15   from emp a
16        join
17        emp b on (a.empno=b.mgr)
18        left join
19        emp c on (b.empno=c.mgr)
20  where a.ename = 'KING'
21 union
22 select concat(a.ename,' - ',b.ename,' - ',
23               c.ename,' - ',d.ename)
24   from emp a
25        join
26        emp b on (a.empno=b.mgr)
27        join
28        emp c on (b.empno=c.mgr)
29        left join
30        emp d on (c.empno=d.mgr)
31  where a.ename = 'KING'
32        ) x
33  where tree is not null
34  order by 1
```

3. 讨论

DB2 和 SQL Server

首先识别出根节点（员工 KING）所在的行，这就是递归视图 X 中 UNION ALL 的前半部分。然后找出 KING 的下属，以及下属的下属（如果存在的话），这一步需要连接递归视图 X 和 EMP 表。递归操作会逐层遍历全体员工。递归视图 X 返回的结果集如下所示，此时尚未针对输出结果做格式化。

```
with x (ename,empno)
    as (
select cast(ename as varchar(100)),empno
  from emp
 where mgr is null
 union all
select cast(e.ename as varchar(100)),e.empno
  from emp e, x
 where e.mgr = x.empno
)
select ename emp_tree
  from x

EMP_TREE
-----------------
KING
JONES
```

```
SCOTT
ADAMS
FORD
SMITH
BLAKE
ALLEN
WARD
MARTIN
TURNER
JAMES
CLARK
MILLER
```

层次关系中的全部 ENAME 都被找出来了（这一点很有用），但因为没有做格式化，我们看不出哪些是管理者。把每个员工和他的管理者连接起来，就能得到更具可读性的输出结果了。下面的表达式出现在递归视图 X 里 UNION ALL 后半部分的 SELECT 子句中，它能生成所需格式的输出结果。

```
cast(x.ename+','+e.ename as varchar(100))
```

WITH 子句对于这一类问题非常有用，因为层次关系发生变动（例如，叶子节点变成了分支节点）时不需要修改查询。

Oracle

CONNECT BY 子句返回层次结构里的行。START WITH 子句定义根节点所在的行。如果不调用 SYS_CONNECT_BY_PATH 函数，我们会看到所需的各行数据都会被返回（这一点很有用），但因为还没做格式化，因而无法展现行之间的关系。

```
select ename emp_tree
  from emp
 start with mgr is null
connect by prior empno = mgr

EMP_TREE
-----------------
KING
JONES
SCOTT
ADAMS
FORD
SMITH
BLAKE
ALLEN
WARD
MARTIN
TURNER
JAMES
CLARK
MILLER
```

通过使用伪列 LEVEL 和函数 LPAD，我们能看到更清晰的层次关系，最终我们也会更深入地理解为什么 SYS_CONNECT_BY_PATH 能够返回我们所需的结果集。

```
select lpad('.',2*level,'.')||ename emp_tree
   from emp
  start with mgr is null
connect by prior empno = mgr

EMP_TREE
------------------
..KING
....JONES
......SCOTT
........ADAMS
......FORD
........SMITH
....BLAKE
......ALLEN
......WARD
......MARTIN
......TURNER
......JAMES
....CLARK
......MILLER
```

以上输出结果里的缩进距离表明了谁是管理者，他们的下属都会被嵌套进更深层的缩进里去。例如，KING 没有上级管理者。JONES 的管理者是 KING。SCOTT 的管理者是 JONES。ADAMS 的管理者是 SCOTT。

如果仔细观察 SYS_CONNECT_BY_PATH 函数连接的行，我们就会明白是 SYS_CONNECT_BY_PATH 展现出了层次关系。有了 SYS_CONNECT_BY_PATH，每当到达一个新节点，我们同时也能看到它的所有父节点。

```
KING
KING - JONES
KING - JONES - SCOTT
KING - JONES - SCOTT – ADAMS
```



对于 Oracle 8*i* 或更早版本，不妨使用 PostgreSQL 解决方案。除此之外，因为更早版本的 Oracle 其实已经支持 CONNECT BY 子句，只要使用 LEVEL 和 RPAD/LPAD 格式化输出结果即可。（当然，模仿 SYS_ CONNECT_BY_PATH 的输出格式要麻烦一些。）

PostgreSQL 和 MySQL

PostgreSQL 和 MySQL 解决方案除了 SELECT 子句里的字符串连接操作不同之外，其余部分都是一样的。首先要决定一个分支里最多会有多少个节点。在编写查询语句之前，我们必须手动完成这个计算。仔细观察 EMP 表的数据，我们会发现员工 ADAM 和 SMITH 是层次最深的叶子节点。（不妨参考 Oracle 解决方案的“讨论”部分，那里面已经用很美观的格式打印出了 EMP 表的层次关系。）我们来看一下员工 ADAMS，ADAMS 的管理者是 SCOTT，SCOTT 的管理者是 JONES，而 JONES 的管理者则是 KING，因此一共 4 层。为了展示 4 层的层次关系，我们必须一口气做 3 次 EMP 表的自连接查询，并且必须写一个包含有 4 个部分的 UNION 查询。下面展示了该自连接查询（即本解决方案中最下面那个

UNION 后面的查询）的结果。（这里使用的是 PostgreSQL 的语法，MySQL 用户需要把字符串连接操作符 || 改为 CONCAT 函数调用。）

```
select rtrim(a.ename||' - '||b.ename||' - '||
             c.ename||' - '||d.ename,' - ') as max_depth_4
  from emp a
       join
       emp b on (a.empno=b.mgr)
       join
       emp c on (b.empno=c.mgr)
       left join
       emp d on (c.empno=d.mgr)
 where a.ename = 'KING'

MAX_DEPTH_4
-----------------------------
KING - JONES - FORD - SMITH
KING - JONES - SCOTT - ADAMS
KING - BLAKE - TURNER
KING - BLAKE - ALLEN
KING - BLAKE - WARD
KING - CLARK - MILLER
KING - BLAKE - MARTIN
KING - BLAKE - JAMES
```

A.ENAME 过滤条件是必须的，因为要确保根节点所在的行是 KING，而不是其他人。如果仔细观察以上结果集，并与最终结果集相比较的话，我们会发现某些 3 层关系的数据没有被打印出来：KING-JONES-FORD 和 KING-JONES-SCOTT。为了使最终结果集包含这些数据，我们需要再写一个类似于上述查询的语句，但要减少一个自连接（做 2 次自连接而不是 3 次）。该查询的结果集如下所示。

```
select rtrim(a.ename||' - '||b.ename
                    ||' - '||c.ename,' - ') as max_depth_3
  from emp a
       join
       emp b on (a.empno=b.mgr)
       left join
       emp c on (b.empno=c.mgr)
 where a.ename = 'KING'

MAX_DEPTH_3
---------------------------
KING - BLAKE - ALLEN
KING - BLAKE - WARD
KING - BLAKE - MARTIN
KING - JONES - SCOTT
KING - BLAKE - TURNER
KING - BLAKE - JAMES
KING - JONES - FORD
KING - CLARK - MILLER
```

像前一个查询一样，上述 A.ENAME 过滤条件也是必须的，因为要确保根节点是 KING。你可能会注意到这两个查询结果中会有一些重叠的行。为了剔除这些多余的行，只要把两个

查询 UNION 起来即可。

```
select rtrim(a.ename||' - '||b.ename
                   ||' - '||c.ename,' - ') as partial_tree
  from emp a
       join
       emp b on (a.empno=b.mgr)
       left join
       emp c on (b.empno=c.mgr)
 where a.ename = 'KING'
union
select rtrim(a.ename||' - '||b.ename||' - '||
             c.ename||' - '||d.ename,' - ')
  from emp a
       join
       emp b on (a.empno=b.mgr)
       join
       emp c on (b.empno=c.mgr)
       left join
       emp d on (c.empno=d.mgr)
 where a.ename = 'KING'

PARTIAL_TREE
----------------------------
KING - BLAKE - ALLEN
KING - BLAKE - JAMES
KING - BLAKE - MARTIN
KING - BLAKE - TURNER
KING - BLAKE - WARD
KING - CLARK - MILLER
KING - JONES - FORD
KING - JONES - FORD - SMITH
KING - JONES - SCOTT
KING - JONES - SCOTT - ADAMS
```

现在，整个树形结构已经基本成型。下一步要找出以 KING 为根节点的、两层关系的行（即 KING 的直接下属）。返回这些行的查询如下所示。

```
select a.ename||' - '||b.ename as max_depth_2
  from emp a
       join
       emp b on (a.empno=b.mgr)
 where a.mgr is null

MAX_DEPTH_2
---------------
KING - JONES
KING - BLAKE
KING - CLARK
```

下一步需要把 PARTIAL_TREE 和上面的查询 UNION 起来。

```
select a.ename||' - '||b.ename as partial_tree
  from emp a
       join
```

```
       emp b on (a.empno=b.mgr)
 where a.mgr is null
union
select rtrim(a.ename||' - '||b.ename
                    ||' - '||c.ename,' - ')
  from emp a
       join
       emp b on (a.empno=b.mgr)
       left join
       emp c on (b.empno=c.mgr)
 where a.ename = 'KING'
union
select rtrim(a.ename||' - '||b.ename||' - '||
             c.ename||' - '||d.ename,' - ')
  from emp a
       join
       emp b on (a.empno=b.mgr)
       join
       emp c on (b.empno=c.mgr)
       left join
       emp d on (c.empno=d.mgr)
 where a.ename = 'KING'

PARTIAL_TREE
----------------------------------
KING - BLAKE
KING - BLAKE - ALLEN
KING - BLAKE - JAMES
KING - BLAKE - MARTIN
KING - BLAKE - TURNER
KING - BLAKE - WARD
KING - CLARK
KING - CLARK - MILLER
KING - JONES
KING - JONES - FORD
KING - JONES - FORD - SMITH
KING - JONES - SCOTT
KING - JONES - SCOTT – ADAMS
```

最后，把 KING 和 PARTIAL_TREE 也 UNION 起来以得到最终结果集。

13.4　找出给定的父节点对应的所有子节点

1. 问题

你想找出 JONES 的下属员工，既包括直接的下属，也包括间接的下属（即这些员工的管理者是 JONES 的下属）。JONES 的所有下属显示如下。（JONES 本人也会包含在该结果集中。）

```
ENAME
---------
JONES
SCOTT
```

```
ADAMS
FORD
SMITH
```

2. 解决方案

能够自由移动到树形结构的顶端或底端是非常有用的一项特性。本解决方案并不要求做特殊的格式化工作。我们的目标只是要显示 JONES 的所有下属，也包括 JONES 本人。这种类型的查询往往能体现出 SQL 递归特性的优势，例如，Oracle 的 CONNECT BY 子句，SQL Server 和 DB2 的 WITH 子句。

DB2 和 SQL Server

使用 WITH 递归查询找出 JONES 的下属。UNION 查询的前半部分，通过指定 WHERE ENAME = 'JONES' 表明递归操作从 JONES 开始。

```
 1    with x (ename,empno)
 2      as (
 3  select ename,empno
 4    from emp
 5   where ename = 'JONES'
 6   union all
 7  select e.ename, e.empno
 8    from emp e, x
 9   where x.empno = e.mgr
10  )
11  select ename
12    from x
```

Oracle

使用 CONNECT BY 子句，并指定 START WITH ENAME = 'JONES' 找出 JONES 的所有下属。

```
1  select ename
2    from emp
3   start with ename = 'JONES'
4 connect by prior empno = mgr
```

PostgreSQL 和 MySQL

我们必须先计算出树形结构包含了多少个节点，下面的查询展示了如何确定层次关系的深度。

```
/* 找到JONES的EMPNO */
select ename,empno,mgr
  from emp
 where ename = 'JONES'

ENAME           EMPNO        MGR
---------- ---------- ----------
JONES            7566       7839

/* JONES有直接下属吗？ */
select count(*)
  from emp
 where mgr = 7566
```

```
  COUNT(*)
----------
         2

/* JONES有两个直接下属,找到他们的EMPNO */
select ename,empno,mgr
  from emp
 where mgr = 7566

ENAME           EMPNO        MGR
---------- ---------- ----------
SCOTT            7788       7566
FORD             7902       7566

/* SCOTT和FORD有下属吗？ */
select count(*)
  from emp
 where mgr in (7788,7902)

  COUNT(*)
----------
         2

/* SCOTT和FORD各有一个下属,找出他们的EMPNO */
select ename,empno,mgr
  from emp
 where mgr in (7788,7902)

ENAME           EMPNO        MGR
---------- ---------- ----------
SMITH            7369       7902
ADAMS            7876       7788

/* SMITH和ADAMS有下属吗？ */
select count(*)
  from emp
 where mgr in (7369,7876)

  COUNT(*)
----------
         0
```

该层次关系起始于 JONES，终止于 SMITH 和 ADAMS。它的深度为 3 层。既然已经知道了深度，我们就可以从顶端到底端依次遍历整个层次关系。

首先，2 次自连接 EMP 表。然后转换内嵌视图 X 把 2 行 3 列的数据集变为 6 行 1 列。（对于 PostgreSQL 而言，还有另一种做法，即可以使用 GENERATE_SERIES(1,6) 来代替数据透视表 T100。）

```
1  select distinct
2         case t100.id
3              when 1 then root
4              when 2 then branch
```

```
 5             else        leaf
 6         end as JONES_SUBORDINATES
 7    from (
 8  select a.ename as root,
 9         b.ename as branch,
10         c.ename as leaf
11    from emp a, emp b, emp c
12   where a.ename = 'JONES'
13     and a.empno = b.mgr
14     and b.empno = c.mgr
15         ) x,
16         t100
17   where t100.id <= 6
```

除此之外，我们还可以使用视图，并把视图的结果集 UNION 起来。我们创建了一些视图，如下所示。

```
create view v1
as
select ename,mgr,empno
  from emp
 where ename = 'JONES'

create view v2
as
select ename,mgr,empno
  from emp
 where mgr = (select empno from v1)

create view v3
as
select ename,mgr,empno
  from emp
 where mgr in (select empno from v2)
```

因而，本解决方案将就变成了下面的查询。

```
select ename from v1
 union
select ename from v2
 union
select ename from v3
```

3. 讨论

DB2 和 SQL Server

WITH 递归查询使得本问题解决起来容易多了。WITH 子句的第一部分，即 UNION ALL 的前半部分，返回了员工 JONES 所在的行。我们需要返回 ENAME 以得到员工的名字，并返回 EMPNO 以便基于它做连接查询。UNION ALL 的后半部分基于 EMP.MGR 和 X.EMPNO 做递归的连接查询。该连接条件将一次次地执行下去，直至遍历完整个结果集。

Oracle

START WITH 子句表明查询操作将以 JONES 为根节点。CONNECT BY 子句的条件驱动树形结构的遍历操作，并将一直执行下去，直到条件不再成立才会停止。

PostgreSQL 和 MySQL

这里用到的技巧来自 13.2 节。本解决方案最显而易见的不足是，我们需要提前知道层次关系的深度。

13.5 确认叶子节点、分支节点和根节点

1. 问题

你想确定给定的一行数据是哪种类型的节点：叶子节点、分支节点还是根节点。对于本实例而言，叶子节点代表不是管理者的员工。分支节点表示管理者的员工，同时他也拥有自己的上级管理者。根节点表示没有上级管理者的员工。你希望通过返回 `1`（`TRUE`）或 `0`（`FALSE`）来反映每一行在层次关系中的状态。你希望返回如下所示的结果集。

```
ENAME         IS_LEAF  IS_BRANCH    IS_ROOT
---------- ---------- ---------- ----------
KING                0          0          1
JONES               0          1          0
SCOTT               0          1          0
FORD                0          1          0
CLARK               0          1          0
BLAKE               0          1          0
ADAMS               1          0          0
MILLER              1          0          0
JAMES               1          0          0
TURNER              1          0          0
ALLEN               1          0          0
WARD                1          0          0
MARTIN              1          0          0
SMITH               1          0          0
```

2. 解决方案

`EMP` 表是树形结构，而不是递归层次结构（recursive hierarchy），因为根节点的 `MGR` 值是 `Null`。认识到这一点很重要。如果 `EMP` 表是递归层次结构，那么根节点将会指向自身（即员工 KING的 `MGR` 值将等于他的 `EMPNO`）。 我认为这是违反常理的，因而指定根节点的 `MGR` 值为 `Null`。对于 Oracle 的 `CONNECT BY` 子句和 DB2/SQL Server 的 `WITH` 子句而言，树形结构处理起来更容易，甚至可能比递归层次结构的处理效率更高。如果必须要处理递归层次结构，并且要使用 `CONNECT BY` 或 `WITH` 子句，那么就要小心了：我们可能会掉进死循环里。为防止陷入递归层次结构设置的陷阱，需要编写额外的代码小心地避开它。

DB2、PostgreSQL、MySQL 和 SQL Server

使用 3 个标量子查询，针对每一种节点类型分别计算出正确的“布尔”值（1 或 0）。

```
1  select e.ename,
2         (select sign(count(*)) from emp d
3           where 0 =
4             (select count(*) from emp f
5               where f.mgr = e.empno)) as is_leaf,
6         (select sign(count(*)) from emp d
7           where d.mgr = e.empno
8             and e.mgr is not null) as is_branch,
```

```
 9        (select sign(count(*)) from emp d
10          where d.empno = e.empno
11            and d.mgr is null) as is_root
12    from emp e
13  order by 4 desc,3 desc
```

Oracle

对于 Oracle Database 10*g* 之前的版本，上述标量子查询方案也适用。下面的解决方案利用（Oracle Database 10*g* 新增的）内置函数来帮助我们识别根节点和叶子节点。这些函数分别是 `CONNECT_BY_ROOT` 和 `CONNECT_BY_ISLEAF`。

```
 1  select ename,
 2         connect_by_isleaf is_leaf,
 3         (select count(*) from emp e
 4           where e.mgr = emp.empno
 5             and emp.mgr is not null
 6             and rownum = 1) is_branch,
 7         decode(ename,connect_by_root(ename),1,0) is_root
 8    from emp
 9   start with mgr is null
10 connect by prior empno = mgr
11 order by 4 desc, 3 desc
```

3. 讨论

DB2、PostgreSQL、MySQL 和 SQL Server

本解决方案按照“问题”部分提出的规则确认叶子节点、分支节点和根节点。首先确认一个员工是否是叶子节点。如果当前员工不是管理者（没有下属），那么他就是一个叶子节点。第一个标量子查询 `IS_LEAF` 如下所示。

```
select e.ename,
       (select sign(count(*)) from emp d
         where 0 =
           (select count(*) from emp f
             where f.mgr = e.empno)) as is_leaf
  from emp e
order by 2 desc

ENAME         IS_LEAF
---------- ----------
SMITH               1
ALLEN               1
WARD                1
ADAMS               1
TURNER              1
MARTIN              1
JAMES               1
MILLER              1
JONES               0
BLAKE               0
CLARK               0
FORD                0
SCOTT               0
KING                0
```

IS_LEAF 的输出结果要么是 0，要么是 1，因此要对 COUNT(*) 查询的结果调用 SIGN 函数。否则的话，对于叶子节点而言，我们将得到 14 而不是 1。作为一种替代方案，因为只希望返回 0 或者 1，我们也可以对一个只有一行数据的表做 COUNT(*) 查询。例如：

```
select e.ename,
       (select count(*) from t1 d
         where not exists
           (select null from emp f
             where f.mgr = e.empno)) as is_leaf
  from emp e
order by 2 desc

ENAME         IS_LEAF
---------- ---------
SMITH               1
ALLEN               1
WARD                1
ADAMS               1
TURNER              1
MARTIN              1
JAMES               1
MILLER              1
JONES               0
BLAKE               0
CLARK               0
FORD                0
SCOTT               0
KING                0
```

下一步要找出分支节点。如果一个员工是管理者（有下属），并且他们也碰巧有上级管理者，那么该员工就是一个分支节点。标量子查询 IS_BRANCH 的结果集如下所示。

```
select e.ename,
       (select sign(count(*)) from emp d
         where d.mgr = e.empno
          and e.mgr is not null) as is_branch
  from emp e
 order by 2 desc

ENAME       IS_BRANCH
---------- ---------
JONES               1
BLAKE               1
SCOTT               1
CLARK               1
FORD                1
SMITH               0
TURNER              0
MILLER              0
JAMES               0
ADAMS               0
KING                0
ALLEN               0
MARTIN              0
WARD                0
```

类似地，有必要对 COUNT(*) 查询的结果调用 SIGN 函数。否则，当一个节点是分支节点时，我们有可能得到比 1 大的值。就像标量子查询 IS_LEAF 一样，我们也可以借助一个只有一行数据的表来避免使用 SIGN 函数。下面的解决方案用到了一个只有一行数据的表 T1。

```
select e.ename,
       (select count(*) from t1 t
         where exists (
          select null from emp f
           where f.mgr = e.empno
            and e.mgr is not null)) as is_branch
  from emp e
order by 2 desc

ENAME       IS_BRANCH
---------- ----------
JONES               1
BLAKE               1
SCOTT               1
CLARK               1
FORD                1
SMITH               0
TURNER              0
MILLER              0
JAMES               0
ADAMS               0
KING                0
ALLEN               0
MARTIN              0
WARD                0
```

最后要找到根节点。根节点的员工是一个管理者，但他没有上级管理者。在 EMP 表里，只有 KING 是根节点。标量子查询 IS_ROOT 如下所示。

```
select e.ename,
       (select sign(count(*)) from emp d
         where d.empno = e.empno
          and d.mgr is null) as is_root
  from emp e
 order by 2 desc

ENAME         IS_ROOT
---------- ----------
KING                1
SMITH               0
ALLEN               0
WARD                0
JONES               0
TURNER              0
JAMES               0
MILLER              0
FORD                0
ADAMS               0
MARTIN              0
BLAKE               0
```

```
CLARK              0
SCOTT              0
```

EMP 表只有 14 行数据，很容易就能看出来 KING 是唯一的根节点，因此严格地说，上述对 COUNT(*) 查询结果调用 SIGN 函数的操作并不是必须的。如果可能存在多个根节点，那么就有必要使用 SIGN 函数。当然，也可以像前面的 IS_BRANCH 和 IS_LEAF 一样，借助一个只有一行数据的表来避免使用 SIGN 函数。

Oracle

对于 Oracle Database 10*g* 之前的版本，可以参考其他数据库的讨论内容，它们的解决方案也适用于 Oracle（无须任何改动）。对于 Oracle 10*g* 和后续的版本，我们可以利用两个更便于识别根节点和叶子节点的函数，它们分别是 CONNECT_BY_ROOT 和 CONNECT_BY_ISLEAF。在写作本书时，若要使用 CONNECT_BY_ROOT 和 CONNECT_BY_ISLEAF 函数，则 SQL 语句中也要同时用到 CONNECT BY 子句。首先，调用 CONNECT_BY_ISLEAF 找出叶子节点，如下所示。

```
select ename,
        connect_by_isleaf is_leaf
  from emp
 start with mgr is null
connect by prior empno = mgr
order by 2 desc

ENAME         IS_LEAF
---------- ----------
ADAMS               1
SMITH               1
ALLEN               1
TURNER              1
MARTIN              1
WARD                1
JAMES               1
MILLER              1
KING                0
JONES               0
BLAKE               0
CLARK               0
FORD                0
SCOTT               0
```

下一步要使用标量子查询找出分支节点。分支节点的员工既是一个管理者，同时也有上级管理者。

```
select ename,
        (select count(*) from emp e
          where e.mgr = emp.empno
            and emp.mgr is not null
            and rownum = 1) is_branch
  from emp
 start with mgr is null
connect by prior empno = mgr
order by 2 desc
```

```
ENAME        IS_BRANCH
---------- ----------
JONES               1
SCOTT               1
BLAKE               1
FORD                1
CLARK               1
KING                0
MARTIN              0
MILLER              0
JAMES               0
TURNER              0
WARD                0
ADAMS               0
ALLEN               0
SMITH               0
```

上述 ROWNUM 过滤条件是必要的，这是为了确保返回值是 1 或 0，而不是其他值。

最后，使用函数 CONNECT_BY_ROOT 识别出根节点。本解决方案找出根节点的 ENAME，并逐一地比较 ENAME 和下面查询返回的行数据。如果两个 ENAME 相等，则那一行即为根节点。

```
select ename,
       decode(ename,connect_by_root(ename),1,0) is_root
  from emp
 start with mgr is null
connect by prior empno = mgr
order by 2 desc

ENAME          IS_ROOT
---------- ----------
KING                1
JONES               0
SCOTT               0
ADAMS               0
FORD                0
SMITH               0
BLAKE               0
ALLEN               0
WARD                0
MARTIN              0
TURNER              0
JAMES               0
CLARK               0
MILLER              0
```

对于 Oracle 9*i* 及其后续版本，也可以使用 SYS_CONNECT_BY_PATH 函数替代 CONNECT_BY_ROOT。Oracle 9*i* 版本的解决方案如下所示。

```
select ename,
       decode(substr(root,1,instr(root,',')-1),NULL,1,0) root
  from (
select ename,
       ltrim(sys_connect_by_path(ename,','),',') root
  from emp
```

```
start with mgr is null
connect by prior empno=mgr
       )

ENAME       ROOT
---------- ----
KING          1
JONES         0
SCOTT         0
ADAMS         0
FORD          0
SMITH         0
BLAKE         0
ALLEN         0
WARD          0
MARTIN        0
TURNER        0
JAMES         0
CLARK         0
MILLER        0
```

SYS_CONNECT_BY_PATH 函数从根节点开始逐一遍历树形结构，如下所示。

```
select ename,
       ltrim(sys_connect_by_path(ename,','),',') path
  from emp
start with mgr is null
connect by prior empno=mgr

ENAME       PATH
---------- ----------------------------
KING        KING
JONES       KING,JONES
SCOTT       KING,JONES,SCOTT
ADAMS       KING,JONES,SCOTT,ADAMS
FORD        KING,JONES,FORD
SMITH       KING,JONES,FORD,SMITH
BLAKE       KING,BLAKE
ALLEN       KING,BLAKE,ALLEN
WARD        KING,BLAKE,WARD
MARTIN      KING,BLAKE,MARTIN
TURNER      KING,BLAKE,TURNER
JAMES       KING,BLAKE,JAMES
CLARK       KING,CLARK
MILLER      KING,CLARK,MILLER
```

为了得到根节点，只要提取出 PATH 里第一个 ENAME 即可。

```
select ename,
       substr(root,1,instr(root,',')-1) root
  from (
select ename,
       ltrim(sys_connect_by_path(ename,','),',') root
  from emp
start with mgr is null
```

```
connect by prior empno=mgr
       )

ENAME       ROOT
---------- ---------
KING
JONES      KING
SCOTT      KING
ADAMS      KING
FORD       KING
SMITH      KING
BLAKE      KING
ALLEN      KING
WARD       KING
MARTIN     KING
TURNER     KING
JAMES      KING
CLARK      KING
MILLER     KING
```

最后，标出 ROOT 列为 Null 的行，它就是我们要寻找的根节点行。

第 14 章

杂项

本章的实例之所以没有被分别放入前几章，不仅因为相关的章节太长了，还因为这些问题本身非常有趣。虽然这些实例的实用性可能不是那么强，我还是希望把本章的内容写得尽量有趣些。总之，我觉得这些实例都很有趣，因此希望把它们包含进本书。

14.1 使用SQL Server的PIVOT操作符创建交叉报表

1. 问题

你想创建一个交叉报表，以实现把行形式的结果集转换为列形式。你已经掌握了传统的做法，但这次希望尝试一些不同的技巧。尤其是你希望不使用 CASE 表达式或连接操作就能返回如下所示的结果集。

```
DEPT_10    DEPT_20    DEPT_30    DEPT_40
------- ---------- ---------- ----------
      3          5          6          0
```

2. 解决方案

使用 PIVOT 操作符生成要求的结果集，而不使用 CASE 表达式或额外的连接操作。

```
1  select [10] as dept_10,
2         [20] as dept_20,
3         [30] as dept_30,
4         [40] as dept_40
5    from (select deptno, empno from emp) driver
6   pivot (
7      count(driver.empno)
```

```
8      for driver.deptno in ( [10],[20],[30],[40] )
9   ) as empPivot
```

3. 讨论

最初看到 PIVOT 操作符可能会觉得陌生，其实在上述解决方案中，它执行的操作在技术上等价于我们所熟知的形式变换查询，如下所示。

```
select sum(case deptno when 10 then 1 else 0 end) as dept_10,
       sum(case deptno when 20 then 1 else 0 end) as dept_20,
       sum(case deptno when 30 then 1 else 0 end) as dept_30,
       sum(case deptno when 40 then 1 else 0 end) as dept_40
  from emp

DEPT_10    DEPT_20    DEPT_30    DEPT_40
------- ---------- ---------- ----------
      3          5          6          0
```

在对 PIVOT 操作有了基本的了解后，现在我们把它拆解开来，看看它究竟做了些什么。上述解决方案中的第 5 行展示了内嵌视图 DRIVER。

```
from (select deptno, empno from emp) driver
```

我选择使用别名 driver，是因为该内嵌视图［即“表表达式”（table expression）］中的数据会直接流入 PIVOT 操作。PIVOT 操作符通过评估第 8 行的 FOR 列表中的项目把行转换为列，如下所示。

```
for driver.deptno in ( [10],[20],[30],[40] )
```

执行过程大致如下。

(1) 如果 DEPTNO 等于 10，就针对相关的行执行预先定义好的聚合操作（COUNT(DRIVER.EMPNO)）。
(2) 针对 DEPTNO 等于 20、30 和 40 的行重复同样的操作。

第 8 行方括号里的项目不仅能够为聚合操作提供值，同时，这些项目也变成了结果集里的列名（不带方括号）。上述解决方案的 SELECT 子句也引用了 FOR 列表里的那些项目，并为它们分别指定了别名。如果不为 FOR 列表里的项目指定别名，则那些项目就会变成列名，但是会去掉方括号。

还有一个非常有趣之处，由于 DRIVER 只是一个普通的内嵌视图，因此可以使用更复杂的 SQL。假设我们希望修改结果集，把实际的部门名称作为结果集的列名。下面列出了 DEPT 表的数据。

```
select * from dept

DEPTNO DNAME          LOC
------ -------------- -------------
    10 ACCOUNTING     NEW YORK
    20 RESEARCH       DALLAS
    30 SALES          CHICAGO
    40 OPERATIONS     BOSTON
```

我们希望借助 PIVOT 返回如下所示的结果集。

```
ACCOUNTING   RESEARCH      SALES OPERATIONS
---------- ---------- ---------- ----------
         3          5          6          0
```

内嵌视图 DRIVER 事实上能够接受任何有效的表表达式，因此可以先把 EMP 表和 DEPT 表连接起来，然后使用 PIVOT 逐一评估其查询结果。下面的查询将返回上述要求的结果集。

```
select [ACCOUNTING] as ACCOUNTING,
       [SALES]      as SALES,
       [RESEARCH]   as RESEARCH,
       [OPERATIONS] as OPERATIONS
  from (
          select d.dname, e.empno
            from emp e,dept d
           where e.deptno=d.deptno

       ) driver
 pivot (
  count(driver.empno)
  for driver.dname in ([ACCOUNTING],[SALES],[RESEARCH],[OPERATIONS])
 ) as empPivot
```

如上所述，PIVOT 提供了一种有趣的转换结果集的办法。如果你以前习惯使用传统的转换方法，现在不妨将其作为另一种选择放入你的工具箱。

14.2 使用SQL Server的UNPIVOT操作符逆向转换交叉报表

1. 问题

你有一个格式良好的结果集，即“宽表”（fat table），你想针对该结果集进行逆向转换。例如，你希望把 1 行 4 列的结果集变成 4 行 2 列。对于 14.1 节中的结果集：

```
ACCOUNTING   RESEARCH      SALES OPERATIONS
---------- ---------- ---------- ----------
         3          5          6          0
```

你想把它转换成下面这样。

```
DNAME                 CNT
-------------- ----------
ACCOUNTING              3
RESEARCH                5
SALES                   6
OPERATIONS              0
```

2. 解决方案

你应该猜到 SQL Server 除了支持 PIVOT 外，也同时支持 UNPIVOT。为了实现结果的逆向转换，只要把前一个实例的查询作为 driver，并把剩下的工作交给 UNPIVOT 操作符即可。我们只需要指定列名。

```
 1  select DNAME, CNT
 2    from (
 3      select [ACCOUNTING] as ACCOUNTING,
 4             [SALES]      as SALES,
 5             [RESEARCH]   as RESEARCH,
 6             [OPERATIONS] as OPERATIONS
 7        from (
 8                select d.dname, e.empno
 9                  from emp e,dept d
10                 where e.deptno=d.deptno
11
12             ) driver
13       pivot (
14         count(driver.empno)
15         for driver.dname in ([ACCOUNTING],[SALES],[RESEARCH],[OPERATIONS])
16       ) as empPivot
17  ) new_driver
18  unpivot (cnt for dname in (ACCOUNTING,SALES,RESEARCH,OPERATIONS)
19  ) as un_pivot
```

但愿你在此之前已经阅读过前一个实例，因为内嵌视图 NEW_DRIVER 直接使用了前一个实例的代码。（如果你不理解该代码，请先查阅前一个实例。）由于你已经理解了第 3 ～ 16 行的代码，唯一陌生的语法就是第 18 行的 UNPIVOT。

UNPIVOT 命令会查看来自 NEW_DRIVER 的结果集，并评估每一行和每一列。例如，UNPIVOT 操作符会评估 NEW_DRIVER 返回的列名。如果等于 ACCOUNTING，就把列名 ACCOUNTING 转换为一个行值（放置于 DNAME 列）。它也会从 NEW_DRIVER 中提取出 ACCOUNTING 的值（该值等于 3），并将其转换为 ACCOUNTING 行的一部分（放置于 CNT 列）。UNPIVOT 会针对 FOR 列表中指定的每个项目执行类似的操作，并把每一项都转换为一行记录。

新的结果集变得“窄”了许多，只有 DNAME 列和 CNT 列，并且有 4 行数据。

```
select DNAME, CNT
  from (
    select [ACCOUNTING] as ACCOUNTING,
           [SALES]      as SALES,
           [RESEARCH]   as RESEARCH,
           [OPERATIONS] as OPERATIONS
      from (
              select d.dname, e.empno
                from emp e,dept d
               where e.deptno=d.deptno

           ) driver
     pivot (
       count(driver.empno)
       for driver.dname in ( [ACCOUNTING],[SALES],[RESEARCH],[OPERATIONS] )
     ) as empPivot
) new_driver
unpivot (cnt for dname in (ACCOUNTING,SALES,RESEARCH,OPERATIONS)
) as un_pivot
```

```
DNAME                CNT
-------------- ---------
ACCOUNTING             3
RESEARCH               5
SALES                  6
OPERATIONS             0
```

14.3 使用Oracle的MODEL子句变换结果集

1. 问题

就像本章第一个实例一样，你希望于熟知的常规变换技巧之外另辟蹊径。你想尝试一下 Oracle 的 MODEL 子句。不同于 SQL Server 的 PIVOT 操作符，Oracle 的 MODEL 子句并不是用来做结果集变换的。说得准确一些，用 MODEL 子句做结果集变换其实算是误用，它并不符合 MODEL 子句的设计意图。尽管如此，MODEL 子句还是为常见问题提供了一种有趣的思路。对于本例而言，你希望把下面的结果集：

```
select deptno, count(*) cnt
  from emp
 group by deptno

DEPTNO        CNT
------ ----------
    10          3
    20          5
    30          6
```

做如下转换。

```
       D10        D20        D30
---------- ---------- ----------
         3          5          6
```

2. 解决方案

和常规的变换技巧一样，要在 MODEL 子句中使用聚合和 CASE 表达式。不同之处在于我们使用数组来存储聚合运算的值，并返回结果集里的数组。

```
select max(d10) d10,
       max(d20) d20,
       max(d30) d30
  from (
select d10,d20,d30
  from ( select deptno, count(*) cnt from emp group by deptno )
 model
  dimension by(deptno d)
   measures(deptno, cnt d10, cnt d20, cnt d30)
   rules(
     d10[any] = case when deptno[cv()]=10 then d10[cv()] else 0 end,
     d20[any] = case when deptno[cv()]=20 then d20[cv()] else 0 end,
     d30[any] = case when deptno[cv()]=30 then d30[cv()] else 0 end
  )
  )
```

3. 讨论

功能强大的 MODEL 子句是对 Oracle SQL 工具箱非常有益的补充。一旦开始使用 MODEL，我们一定会被它提供的各种实用的功能吸引住。比如迭代，以数组形式访问行值，向结果集 upsert[1] 行值的能力，以及构建参考模型的能力。虽然本实例不会用到 MODEL 子句的这些非常酷的功能，但（出于学习和研究的目的）从多种角度审视问题，并以出人意料的方式使用某些功能未尝不是一次有益的尝试。

理解本解决方案的第一步是，仔细观察 FROM 子句的内嵌视图。该内嵌视图用于统计 EMP 表中每个 DEPTNO 对应的员工总数。结果集显示如下。

```
select deptno, count(*) cnt
  from emp
 group by deptno

DEPTNO        CNT
------ ----------
    10          3
    20          5
    30          6
```

以上结果集就是 MODEL 要处理的数据。仔细观察 MODEL 子句，可以发现它有 3 个组成部分：DIMENSION BY、MEASURES 和 RULES。先从 MEASURES 开始。

MEASURES 列表的项目就是我们为该查询声明的数组。该查询使用了 4 个数组：DEPTNO、D10、D20 和 D30。与 SELECT 列表类似，MEASURES 列表里的数组也可以指定别名。不难发现，上述 4 个数组中有 3 个实际上都来源于内嵌视图的 CNT。

如果说 MEASURES 列表包含了我们用到的数组，那么 DIMENSION BY 子句包含的项目则是数组的索引。试想一下，数组 D10 只是 CNT 的别名。再看一下上述内嵌视图的结果集，我们会发现 CNT 有 3 个值：3、5 和 6。当基于 CNT 值创建一个数组时，我们创建的是拥有 3 个元素的数组，即 3 个整数值：3、5 和 6。现在，我们该如何逐一地访问该数组的值呢？需要借助数组索引。由 DIMENSION BY 子句定义的索引值如下：10、20 和 30（来自上述结果集）。因此，以下面的表达式为例。

```
d10[10]
```

该表达式的评估结果为 3，该值就是数组 D10 中 DEPTNO 10 对应的 CNT 值（该值为 3）。

3 个数组（D10、D20、D30）都是基于 CNT 值创建的，因此它们都有相同的值。那么，如何把合适的统计值放入到正确的数组中呢？这是 RULES 子句要做的事。如果仔细观察上述内嵌视图的结果集的话，我们会看到 DEPTNO 的值分别为 10、20 和 30。RULES 子句里的 CASE 表达式只要逐一评估 DEPTNO 数组的每个值即可。

- 如果值为 10，为 D10[10] 存入 DEPTNO 10 对应的 CNT 值，否则存入 0。
- 如果值为 20，为 D20[20] 存入 DEPTNO 20 对应的 CNT 值，否则存入 0。
- 如果值为 30，为 D30[30] 存入 DEPTNO 30 对应的 CNT 值，否则存入 0。

注 1：“upsert” 的原意是 “如果表中已经存在相关记录，则执行 UPDATE 操作；否则，执行 INSERT 操作”，但此处指的是对中间查询结果进行修改的操作，而不是针对物理上存在的表。——译者注

如果你感到迷惑不解，请不要担心。我们接下来不妨先执行一下到目前为止讨论过的查询代码，下面是刚刚讨论过的查询代码对应的结果集。有时候读一段文字内容，然后实际执行一下对应的代码，最后再回过头去重新读一遍文字内容，这样理解起来会更容易。实际执行一下下面的代码之后，相信你就能迅速理解到目前为止我们讲过的内容。

```
select deptno, d10,d20,d30
  from ( select deptno, count(*) cnt from emp group by deptno )
 model
  dimension by(deptno d)
   measures(deptno, cnt d10, cnt d20, cnt d30)
   rules(
     d10[any] = case when deptno[cv()]=10 then d10[cv()] else 0 end,
     d20[any] = case when deptno[cv()]=20 then d20[cv()] else 0 end,
     d30[any] = case when deptno[cv()]=30 then d30[cv()] else 0 end
  )

DEPTNO        D10        D20        D30
------ ---------- ---------- ----------
    10          3          0          0
    20          0          5          0
    30          0          0          6
```

如上所示，正是 RULES 子句改变了每一个数组里的值。如果你仍然感到疑惑不解，不妨实际执行一下如下所示的查询语句，该查询注释掉了 RULES 子句里的表达式，其他部分与上述查询相同。

```
select deptno, d10,d20,d30
  from ( select deptno, count(*) cnt from emp group by deptno )
 model
  dimension by(deptno d)
   measures(deptno, cnt d10, cnt d20, cnt d30)
   rules(
    /*
     d10[any] = case when deptno[cv()]=10 then d10[cv()] else 0 end,
     d20[any] = case when deptno[cv()]=20 then d20[cv()] else 0 end,
     d30[any] = case when deptno[cv()]=30 then d30[cv()] else 0 end
    */
  )

DEPTNO        D10        D20        D30
------ ---------- ---------- ----------
    10          3          3          3
    20          5          5          5
    30          6          6          6
```

现在应该足够清楚明白了，上述 MODEL 子句的结果集和内嵌视图完全相同，只是 COUNT 函数的返回值被分别指定了别名 D10、D20 和 D30。如下所示的查询也能证明这一点。

```
select deptno, count(*) d10, count(*) d20, count(*) d30
  from emp
 group by deptno
```

```
DEPTNO        D10        D20        D30
------ ---------- ---------- ----------
    10          3          3          3
    20          5          5          5
    30          6          6          6
```

因此，MODEL 子句所做的事情就是取出 DEPTNO 和 CNT 的值，把它们放入数组，并确保每一个数组代表一个单独的 DEPTNO。现在，每一个数组 D10、D20 和 D30 都含有一个非零值，它们代表给定 DEPTNO 对应的 CNT。结果集变换已经完成，剩下的就是调用聚合函数 MAX 以返回单独的一行结果。（本书前几章已经多次使用 MIN 或 SUM，本实例的做法也出于同样的目的。）

```
select max(d10) d10,
       max(d20) d20,
       max(d30) d30
  from (
select d10,d20,d30
  from ( select deptno, count(*) cnt from emp group by deptno )
 model
  dimension by(deptno d)
   measures(deptno, cnt d10, cnt d20, cnt d30)
   rules(
    d10[any] = case when deptno[cv()]=10 then d10[cv()] else 0 end,
    d20[any] = case when deptno[cv()]=20 then d20[cv()] else 0 end,
    d30[any] = case when deptno[cv()]=30 then d30[cv()] else 0 end
  )
  )

       D10        D20        D30
---------- ---------- ----------
         3          5          6
```

14.4　从不固定位置提取字符串的元素

1. 问题

你有一个字符串，其中包含一段连续的日志数据。你想解析该字符串，并从中提取出部分信息。不过，你需要的信息并不存在于字符串的固定位置。因此，你必须借助目标信息附近的某些字符来定位并提取所需的内容。例如，考虑下面的字符串。

```
xxxxxabc[867]xxx[-]xxxx[5309]xxxxx
xxxxxtime:[11271978]favnum:[4]id:[Joe]xxxxx
call:[F_GET_ROWS( )]b1:[ROSEWOOD...SIR]b2:[44400002]77.90xxxxx
film:[non_marked]qq:[unit]tailpipe:[withabanana?]80sxxxxx
```

你希望提取出方括号内的值，返回如下所示的结果集。

```
FIRST_VAL       SECOND_VAL      LAST_VAL
--------------- --------------- ---------------
867             -               5309
11271978        4               Joe
F_GET_ROWS( )   ROSEWOOD...SIR  44400002
non_marked      unit            withabanana?
```

2. 解决方案

尽管不知道我们所感兴趣的字符的确切位置，但我们确定它们是被包含在方括号“[]”中的，并且知道有 3 组这样的值。使用 Oracle 的内置函数 INSTR 找出方括号的位置。使用内置函数 SUBSTR 从字符串中提取所需要的值。视图 V 中包含了我们要解析的字符串，它的定义如下所示。（它的存在只是为了增强代码的可读性。）

```
create view V
as
select 'xxxxxabc[867]xxx[-]xxxx[5309]xxxxx' msg
  from dual
 union all
select 'xxxxxtime:[11271978]favnum:[4]id:[Joe]xxxxx' msg
  from dual
 union all
select 'call:[F_GET_ROWS()]b1:[ROSEWOOD...SIR]b2:[44400002]77.90xxxxx' msg
  from dual
 union all
select 'film:[non_marked]qq:[unit]tailpipe:[withabanana?]80sxxxxx' msg
  from dual
```

```
 1  select substr(msg,
 2          instr(msg,'[',1,1)+1,
 3          instr(msg,']',1,1)-instr(msg,'[',1,1)-1) first_val,
 4         substr(msg,
 5          instr(msg,'[',1,2)+1,
 6          instr(msg,']',1,2)-instr(msg,'[',1,2)-1) second_val,
 7         substr(msg,
 8          instr(msg,'[',-1,1)+1,
 9          instr(msg,')',-1,1)-instr(msg,'[',-1,1]-1) last_val
10    from V
```

3. 讨论

有了 Oracle 的内置函数 INSTR，很容易就能解决本问题。由于已经知道我们感兴趣的值被“[]”包围，并且有 3 组“[]”，那么，本解决方案的第一步就是要使用 INSTR 在每个字符串中找出“[]”的位置。下面的例子返回了每一行中 3 个左方括号和 3 个右方括号的确切位置。

```
select instr(msg,'[',1,1) “1st_[“,
       instr(msg,']',1,1) “]_1st”,
       instr(msg,'[',1,2) “2nd_[“,
       instr(msg,']',1,2) “]_2nd”,
       instr(msg,'[',-1,1) “3rd_[“,
       instr(msg,')',-1,1) “]_3rd”
  from V

 1st_[ ]_1st      2nd_[ ]_2nd      3rd_[ ]_3rd
------ ----- ---------- ----- ---------- -----
     9    13         17    19         24    29
    11    20         28    30         34    38
     6    19         23    38         42    51
     6    17         21    26         36    49
```

现在，最困难的工作已经完成了。剩下的只需要把方括号的位置插入到 SUBSTR 以实现 MSG 的解析即可。你可能已经注意到上述完整的解决方案里有一些针对 INSTR 返回值的简单的数学运算，例如：+1 和 -1；这是为了确保左方括号 [不会被返回。相较于完整的解决方案，下面给出的查询语句不包括这些 +1 和 -1 的操作。注意，每一个返回值开头的字符都是左方括号。

```
select substr(msg,
       instr(msg,'[',1,1],
       instr(msg,']',1,1)-instr(msg,'[',1,1]) first_val,
      substr(msg,
       instr(msg,'[',1,2],
       instr(msg,']',1,2)-instr(msg,'[',1,2]) second_val,
      substr(msg,
       instr(msg,'[',-1,1],
       instr(msg,']',-1,1)-instr(msg,'[',-1,1]) last_val
  from V

FIRST_VAL       SECOND_VAL      LAST_VAL
--------------- --------------- -------------
[867            [-              [5309
[11271978       [4              [Joe
[F_GET_ROWS()   [ROSEWOOD...SIR [44400002
[non_marked     [unit           [withabanana?
```

从以上结果集中可以看到，左方括号也被返回了。你可能会想：“好吧，为 INSTR 的返回值加上 1，这样就去掉了左方括号。为什么要减去 1 呢？”因为如果我们把加 1 的操作放回去，而不添加减 1 操作的话，右方括号就会被返回，如下所示。

```
select substr(msg,
       instr(msg,'[',1,1]+1,
       instr(msg,']',1,1)-instr(msg,'[',1,1)) first_val,
      substr(msg,
       instr(msg,'[',1,2]+1,
       instr(msg,']',1,2)-instr(msg,'[',1,2)) second_val,
      substr(msg,
       instr(msg,'[',-1,1]+1,
       instr(msg,')',-1,1)-instr(msg,'[',-1,1]) last_val
  from V

FIRST_VAL       SECOND_VAL      LAST_VAL
--------------- --------------- -------------
867]            -]              5309]
11271978]       4]              Joe]
F_GET_ROWS()]   ROSEWOOD...SIR] 44400002]
non_marked]     unit]           withabanana?]
```

现在应该很清楚了，为了确保不返回方括号，我们必须在索引开始的位置加上 1，并在索引结束的位置减去 1。

14.5　计算一年有多少天

1. 问题

计算一年有多少天。



本实例为 9.2 节补充了新的解法，该解决方案只适用于 Oracle。

2. 解决方案

使用 TO_CHAR 函数把一年的最后一天格式化为用 3 位数字表示的日期序号。

```
1  select 'Days in 2005: '||
2         to_char(add_months(trunc(sysdate,'y'),12)-1,'DDD')
3         as report
4    from dual
5  union all
6  select 'Days in 2004: '||
7         to_char(add_months(trunc(
8                      to_date('01-SEP-2004'),'y'),12)-1,'DDD')
9    from dual

REPORT
-----------------
Days in 2005: 365
Days in 2004: 366
```

3. 讨论

首先调用 TRUNC 函数返回给定日期所属年份的第一天，如下所示。

```
select trunc(to_date('01-SEP-2004'),'y')
  from dual

TRUNC(TO_DA
-----------
01-JAN-2004
```

下一步，调用 ADD_MONTHS 为截取的日期值加上一年（12 个月）。然后，再减去一天，得到最初的给定日期所属年份的最后一天。

```
select add_months(
         trunc(to_date('01-SEP-2004'),'y'),
         12) before_subtraction,
       add_months(
         trunc(to_date('01-SEP-2004'),'y'),
         12)-1 after_subtraction
  from dual

BEFORE_SUBT AFTER_SUBTR
----------- -----------
01-JAN-2005 31-DEC-2004
```

现在已经得到了目标年份的最后一天，接下来只要调用 TO_CHAR 返回一个 3 位数字即可，该数字表示最后一天在这一年中的日期序号（第 1 天、第 50 天，等等）。

```
select to_char(
         add_months(
           trunc(to_date('01-SEP-2004'),'y'),
           12)-1,'DDD') num_days_in_2004
  from dual

NUM
---
366
```

14.6 查找含有数字和字母的字符串

1. 问题

你有一列含有数字和字母的字符串数据，并且希望返回那些既有字母又有数字的行。换句话说，如果一个字符串只含有数字或只含有字母，则不返回。返回值应该既包含字母又包含数字，考虑如下所示的数据。

```
STRINGS
------------
1010 switch
333
3453430278
ClassSummary
findRow 55
threes
```

最终的结果集应该只有那些既含有字母又含有数字的行。

```
STRINGS
------------
1010 switch
findRow 55
```

2. 解决方案

使用内置函数 TRANSLATE 把每一个字母或数字转换成指定的特殊字符，然后只保留那些每种特殊字符至少都出现过一次的字符串。本解决方案使用了 Oracle 句法，但 DB2 和 PostgreSQL 也都支持 TRANSLATE，因此你能够很容易地对本解决方案做出适当改动以适用于另一个数据库。

```
with v as (
select 'ClassSummary' strings from dual union
select '3453430278'           from dual union
select 'findRow 55'           from dual union
select '1010 switch'          from dual union
select '333'                  from dual union
select 'threes'               from dual
)
select strings
```

```
  from (
select strings,
       translate(
         strings,
         'abcdefghijklmnopqrstuvwxyz0123456789',
         rpad('#',26,'#')||rpad('*',10,'*')) translated
  from v
       ) x
 where instr(translated,'#') > 0
   and instr(translated,'*') > 0
```



如果不想使用 WITH 子句，也可以使用内嵌视图或直接创建一个视图。

3. 讨论

有了 TRANSLATE 函数，本问题的解决就非常简单了。首先借助 TRANSLATE 把所有字母和数字分别替换为 # 和 *，中间结果（来自内嵌视图 X）显示如下。

```
with v as (
select 'ClassSummary' strings from dual union
select '3453430278'           from dual union
select 'findRow 55'           from dual union
select '1010 switch'          from dual union
select '333'                  from dual union
select 'threes'               from dual
)
select strings,
       translate(
         strings,
         'abcdefghijklmnopqrstuvwxyz0123456789',
         rpad('#',26,'#')||rpad('*',10,'*')) translated
  from v

STRINGS      TRANSLATED
------------ ------------
1010 switch  **** ######
333          ***
3453430278   **********
ClassSummary C####S######
findRow 55   ####R## **
threes       ######
```

现在，只剩下了一个问题，就是只保留那些 # 和 * 都至少出现过一次的行。使用函数 INSTR 判断一个字符串中是否包含 # 和 *。如果两种字符都出现过，那么返回值将大于 0。为了让你看得更加清楚明白，下面列出了最终返回的字符串和转换后的值。

```
with v as (
select 'ClassSummary' strings from dual union
select '3453430278'           from dual union
select 'findRow 55'           from dual union
select '1010 switch'          from dual union
select '333'                  from dual union
```

```
select 'threes'               from dual
)
select strings, translated
  from (
select strings,
       translate(
        strings,
        'abcdefghijklmnopqrstuvwxyz0123456789',
        rpad('#',26,'#')||rpad('*',10,'*')) translated
  from v
       )
 where instr(translated,'#') > 0
   and instr(translated,'*') > 0

STRINGS      TRANSLATED
----------- -----------
1010 switch  **** ######
findRow 55   ####R## **
```

14.7 在Oracle中把整数转换成二进制

1. 问题

在 Oracle 数据库中，你想把一个整数转换为二进制形式。例如，你希望返回 EMP 表里的所有工资，以二进制形式显示的工资也要作为结果集的一部分被返回，如下所示。

```
ENAME        SAL SAL_BINARY
---------- ----- -------------------
SMITH        800 1100100000
ALLEN       1600 11001000000
WARD        1250 10011100010
JONES       2975 101110011111
MARTIN      1250 10011100010
BLAKE       2850 101100100010
CLARK       2450 100110010010
SCOTT       3000 101110111000
KING        5000 1001110001000
TURNER      1500 10111011100
ADAMS       1100 10001001100
JAMES        950 1110110110
FORD        3000 101110111000
MILLER      1300 10100010100
```

2. 解决方案

本解决方案会用到 MODEL 子句，因此需要运行在 Oracle Database 10*g* 或更新的版本上。MODEL 具有迭代和以数组形式访问行值的能力，选择它来解决本问题是顺理成章的做法。（此处假设我们必须使用 SQL 解决本问题，如果能以函数的形式存在则更加适合。）就像本书的其他实例一样，或许你一时看不到那些解决方案的实际应用场景，但是我建议你重点关注其中展示的技术和技巧。关于 MODEL 子句，我们需要认识到它具有面向过程编程的能力，同时又没有丢掉 SQL 原有的面向集合编程的能力。因此，你可能会说："我绝对不会用 SQL 做这些事。"这也没有关系。实际上，我不会建议你应该这样做或者不应该那样做。

我只是提醒你把注意力放在本解决方案展示的技术和技巧上，这样今后遇到更适当的场景时，它自然而然就会派上用场。

下面给出的解决方案从 EMP 表中取出了全部 ENAME 和 SAL，同时又以标量子查询的形式调用了 MODEL 子句。（该标量子查询的作用类似于一个独立存在的函数，接受输入，启动处理过程，最后返回一个值，非常像一个函数做的事。）

```
 1 select ename,
 2        sal,
 3        (
 4        select bin
 5          from dual
 6         model
 7         dimension by ( 0 attr )
 8         measures ( sal num,
 9                    cast(null as varchar2(30)) bin,
10                    '0123456789ABCDEF' hex
11                  )
12         rules iterate (10000) until (num[0] <= 0) (
13           bin[0] = substr(hex[cv()],mod(num[cv()],2)+1,1)||bin[cv()],
14           num[0] = trunc(num[cv()]/2)
15         )
16        ) sal_binary
17   from emp
```

3. 讨论

我在前面的“解决方案”部分中提到过，上述解决方案最好能以函数形式存在。事实上，本实例的灵感确实来自一个函数。我把一个名为 TO_BASE 的函数做了些改编从而写出了本实例，该函数的作者是 Oracle 公司的 Tom Kyte。就像你不会在实际工作中用到本书的其他许多实例一样，本实例的代码也可能对你没有什么实用价值，但不可否认它很好地展示了 MODEL 子句的部分功能，例如迭代和以数组形式访问行值的能力。

为了便于解释，我对上述包含 MODEL 子句的子查询做了一点轻微的改动。下面的代码基本上完全等同于上述解决方案里的那个子查询，但是我让它仅返回数字 2 的二进制形式。

```
select bin
  from dual
 model
 dimension by ( 0 attr )
 measures ( 2 num,
            cast(null as varchar2(30)) bin,
            '0123456789ABCDEF' hex
          )
 rules iterate (10000) until (num[0] <= 0) (
   bin[0] = substr (hex[cv()],mod(num[cv()],2)+1,1)||bin[cv()],
   num[0] = trunc(num[cv()]/2)
 )

BIN
----------
10
```

下面的查询输出了上述 RULES 迭代一次的返回值。

```
select 2 start_val,
       '0123456789ABCDEF' hex,
       substr('0123456789ABCDEF',mod(2,2)+1,1) ||
       cast(null as varchar2(30)) bin,
       trunc(2/2) num
  from dual

START_VAL HEX             BIN        NUM
--------- --------------- ---------- ---
        2 0123456789ABCDEF 0           1
```

START_VAL 代表我们希望把它变成二进制形式的整数，本例中为 2。BIN 的值是针对 0123456789ABCDEF（别名为 HEX）执行子字符串操作的结果。NUM 值是为了测试何时退出循环。

如上述结果集所示，第一次循环的运算结果 BIN 等于 0，NUM 等于 1。由于 NUM 不满足小于或等于 0 的条件，因而会再循环一次。下面的 SQL 语句展示了下一次迭代的结果。

```
select num start_val,
       substr('0123456789ABCDEF',mod(1,2)+1,1) || bin bin,
       trunc(1/2) num
  from (
select 2 start_val,
       '0123456789ABCDEF' hex,
       substr('0123456789ABCDEF',mod(2,2)+1,1) ||
       cast(null as varchar2(30)) bin,
       trunc(2/2) num
  from dual
       )

START_VAL BIN        NUM
--------- ---------- ---
        1 10           0
```

下一次循环运算，针对 HEX 执行的子字符串操作结果返回了 1，它后面要接上前一次的 BIN 值 0。测试用的 NUM 现在等于 0，因此这会是最后一次迭代，而返回值 10 是数字 2 的二进制表示。如果你能理解截止目前的代码，则不妨去掉 MODEL 子句里的迭代语法，从而可以一步一步地理解最终结果是如何计算出来，如下所示。

```
select 2 orig_val, num, bin
  from dual
 model
 dimension by ( 0 attr )
 measures ( 2 num,
            cast(null as varchar2(30)) bin,
            '0123456789ABCDEF' hex
          )
 rules (
    bin[0] = substr (hex[cv()],mod(num[cv()],2)+1,1)||bin[cv()],
    num[0] = trunc(num[cv()]/2),
    bin[1] = substr (hex[0],mod(num[0],2)+1,1)||bin[0],
```

```
    num[1] = trunc(num[0]/2)
  )

ORIG_VAL NUM BIN
-------- --- ---------
       2   1 0
       2   0 10
```

14.8 变换已排名的结果集

1. 问题

你想为一个表里的记录排名，然后把它变换成一个 3 列的结果集。这 3 列分别是前 3 名，接下来的 3 个名次，然后是其余各行记录。例如，你希望根据 SAL 值为 EMP 表的员工排名，并把结果变换成一个 3 列的结果集。你期望的结果集如下所示。

```
TOP_3          NEXT_3         REST
-------------- -------------- -------------
KING   (5000)  BLAKE  (2850)  TURNER (1500)
FORD   (3000)  CLARK  (2450)  MILLER (1300)
SCOTT  (3000)  ALLEN  (1600)  MARTIN (1250)
JONES  (2975)                 WARD   (1250)
                              ADAMS  (1100)
                              JAMES  (950)
                              SMITH  (800)
```

2. 解决方案

本解决方案的关键是先借助窗口函数 DENSE_RANK OVER 根据 SAL 为员工排名，因为该函数允许 Tie 的存在。有了 DENSE_RANK OVER 函数，我们很容易查看前 3 名的工资，接下来的 3 个次高的工资，以及其余工资。

然后，使用窗口函数 ROW_NUMBER OVER 在每个分组内部（前 3 名、接下来的 3 个名次，或其余各行记录）为员工排序。然后，只要做一次常规的行列变换操作，并借助数据库内置的字符串函数对查询结果做出适当格式化即可。下面的解决方案使用 Oracle 语法。DB2 和 SQL Server 2005 都支持窗口函数，相信你能够适当地修改本解决方案以适用于其他数据库。

```
 1  select max(case grp when 1 then rpad(ename,6) ||
 2                      ' ('|| sal ||')' end) top_3,
 3         max(case grp when 2 then rpad(ename,6) ||
 4                      ' ('|| sal ||')' end) next_3,
 5         max(case grp when 3 then rpad(ename,6) ||
 6                      ' ('|| sal ||')' end) rest
 7    from (
 8  select ename,
 9         sal,
10         rnk,
11         case when rnk <= 3 then 1
12              when rnk <= 6 then 2
13              else                3
14         end grp,
```

```
15         row_number()over (
16           partition by case when rnk <= 3 then 1
17                             when rnk <= 6 then 2
18                             else               3
19                        end
20               order by sal desc, ename
21         ) grp_rnk
22    from (
23  select ename,
24         sal,
25         dense_rank()over(order by sal desc) rnk
26    from emp
27         ) x
28         ) y
29   group by grp_rnk
```

3. 讨论

本实例充分展示了窗口函数的威力。上述解决方案看起来很复杂，但是如果我们把它拆解开来一一细看的话，就会发现其实并不难理解。我们先从内嵌视图 X 开始。

```
select ename,
       sal,
       dense_rank()over(order by sal desc) rnk
  from emp

ENAME        SAL        RNK
---------- ----- ----------
KING        5000          1
SCOTT       3000          2
FORD        3000          2
JONES       2975          3
BLAKE       2850          4
CLARK       2450          5
ALLEN       1600          6
TURNER      1500          7
MILLER      1300          8
WARD        1250          9
MARTIN      1250          9
ADAMS       1100         10
JAMES        950         11
SMITH        800         12
```

如上所示，内嵌视图 X 根据 SAL 为员工排序，同时又允许 Tie 的存在。（因为本解决方案使用的是 DENSE_RANK 函数而不是 RANK 函数，DENSE_RANK 不仅允许 Tie 的存在，还能保证名次连续，中间不留空白。）下一步是从内嵌视图 X 里提取出各行数据，并借助 CASE 表达式评估 DENSE_RANK 返回的排名结果从而实现分组。除此之外，还要调用窗口函数 ROW_NUMBER OVER 根据 SAL 对员工进行组内编号（在前面由 CASE 表达式实现的分组里）。所有这些都由内嵌视图 Y 实现的，如下所示。

```
select ename,
       sal,
       rnk,
       case when rnk <= 3 then 1
```

```
              when rnk <= 6 then 2
              else               3
          end grp,
          row_number()over (
            partition by case when rnk <= 3 then 1
                              when rnk <= 6 then 2
                              else               3
                          end
                order by sal desc, ename
          ) grp_rnk
    from (
  select ename,
         sal,
         dense_rank()over(order by sal desc) rnk
    from emp
         )x

ENAME        SAL  RNK  GRP GRP_RNK
---------- ----- ---- ---- -------
KING        5000    1    1       1
FORD        3000    2    1       2
SCOTT       3000    2    1       3
JONES       2975    3    1       4
BLAKE       2850    4    2       1
CLARK       2450    5    2       2
ALLEN       1600    6    2       3
TURNER      1500    7    3       1
MILLER      1300    8    3       2
MARTIN      1250    9    3       3
WARD        1250    9    3       4
ADAMS       1100   10    3       5
JAMES        950   11    3       6
SMITH        800   12    3       7
```

查询语句越来越接近最终的形态了，并且如果你是从开始部分（从内嵌视图 X）一直读到这里的话，一定会发现其实该查询并不是那么难以理解。上述查询返回了每个员工的 SAL、RNK（代表该员工的 SAL 在全体员工中的排名）、GRP（根据每个员工的 SAL 产生的分组），以及 GRP_RANK（在每一个分组内基于 SAL 生成的编号）。

现在，我们要执行一次传统的行列翻转，同时使用 Oracle 的字符串连接操作符 || 把 SAL 附加在 ENAME 的后面。函数 RPAD 确保圆括号里的数值能够上下对齐。最后，针对 GRP_RNK 执行 GROUP BY 操作以确保结果集能包含所有员工。最终的结果集如下所示。

```
  select max(case grp when 1 then rpad(ename,6) ||
                      ' ('|| sal ||')' end) top_3,
         max(case grp when 2 then rpad(ename,6) ||
                      ' ('|| sal ||')' end) next_3,
         max(case grp when 3 then rpad(ename,6) ||
                      ' ('|| sal ||')' end) rest
    from (
  select ename,
         sal,
         rnk,
```

```
       case when rnk <= 3 then 1
            when rnk <= 6 then 2
            else               3
       end grp,
       row_number()over (
         partition by case when rnk <= 3 then 1
                           when rnk <= 6 then 2
                           else               3
                       end
             order by sal desc, ename
       ) grp_rnk
  from (
select ename,
       sal,
       dense_rank()over(order by sal desc) rnk
  from emp
       ) x
       ) y
 group by grp_rnk

TOP_3           NEXT_3          REST
--------------- --------------- -------------
KING   (5000)   BLAKE  (2850)   TURNER (1500)
FORD   (3000)   CLARK  (2450)   MILLER (1300)
SCOTT  (3000)   ALLEN  (1600)   MARTIN (1250)
JONES  (2975)                   WARD   (1250)
                                ADAMS  (1100)
                                JAMES  (950)
                                SMITH  (800)
```

仔细观察以上每一步的查询语句，我们会发现直接读取 EMP 表的操作其实只发生了一次。窗口函数最为引人注目的功能之一就是，只需访问一次原始数据就可以完成很多复杂的任务。不需要自连接或临时表，只要准备好必要的基础数据集，剩下的工作交给窗口函数处理就行了。我们只需要在内嵌视图 X 中访问一次 EMP 表。此后，只要多次变换该查询结果直到得出我们希望看到的结果集即可。想想我们创建出了这么复杂的报表，却只读取过一次原始数据，真是非常酷。

14.9　为两次变换后的结果集增加列标题

1. 问题

你想把两个结果集叠加起来，并把它们转换成两列。除此之外，你还希望为每一列中的每一组行数据增加一个列标题。例如，你有两个表，它们分别是公司里从事两个不同领域工作的员工名单（假设是研究领域和应用领域）。

```
select * from it_research

DEPTNO ENAME
------ --------------------
   100 HOPKINS
   100 JONES
   100 TONEY
```

```
   200 MORALES
   200 P.WHITAKER
   200 MARCIANO
   200 ROBINSON
   300 LACY
   300 WRIGHT
   300 J.TAYLOR

select * from it_apps

DEPTNO ENAME
------ ----------------
   400 CORRALES
   400 MAYWEATHER
   400 CASTILLO
   400 MARQUEZ
   400 MOSLEY
   500 GATTI
   500 CALZAGHE
   600 LAMOTTA
   600 HAGLER
   600 HEARNS
   600 FRAZIER
   700 GUINN
   700 JUDAH
   700 MARGARITO
```

你希望创建一个报表，分两列列出每个表的员工。你也希望返回每个 DEPTNO 下的 ENAME。最终你想得到如下所示的结果集。

```
RESEARCH             APPS
-------------------- ---------------
100                  400
  JONES                MAYWEATHER
  TONEY                CASTILLO
  HOPKINS              MARQUEZ
200                    MOSLEY
  P.WHITAKER           CORRALES
  MARCIANO           500
  ROBINSON             CALZAGHE
  MORALES              GATTI
300                  600
  WRIGHT               HAGLER
  J.TAYLOR             HEARNS
  LACY                 FRAZIER
                       LAMOTTA
                     700
                       JUDAH
                       MARGARITO
                       GUINN
```

2. 解决方案

基本上，本解决方案只需要一个简单的 stack-n-pivot 操作（先执行 UNION 操作，然后再做行列翻转）。除此之外，还要做一个额外的操作：DEPTNO 必须要先于 ENAME 被返回。这

里使用的技巧是，借助笛卡儿积为每个 `DEPTNO` 产生一行额外的数据，这样我们就不仅得到所有员工的数据，也得到了 `DEPTNO` 对应的行。本解决方案采用 Oracle 语法，但是由于 DB2 的窗口函数也支持滑动窗口（`Framing` 子句），适当修改一下本解决方案不难得到适用于 DB2 的代码。因为 `IT_ RESEARCH` 表和 `IT_APPS` 表只为本实例而存在，下面的解决方案里也顺便列出了创建这些表的语句。

```
create table IT_research (deptno number, ename varchar2(20))

insert into IT_research values (100,'HOPKINS')
insert into IT_research values (100,'JONES')
insert into IT_research values (100,'TONEY')
insert into IT_research values (200,'MORALES')
insert into IT_research values (200,'P.WHITAKER')
insert into IT_research values (200,'MARCIANO')
insert into IT_research values (200,'ROBINSON')
insert into IT_research values (300,'LACY')
insert into IT_research values (300,'WRIGHT')
insert into IT_research values (300,'J.TAYLOR')

create table IT_apps (deptno number, ename varchar2(20))

insert into IT_apps values (400,'CORRALES')
insert into IT_apps values (400,'MAYWEATHER')
insert into IT_apps values (400,'CASTILLO')
insert into IT_apps values (400,'MARQUEZ')
insert into IT_apps values (400,'MOSLEY')
insert into IT_apps values (500,'GATTI')
insert into IT_apps values (500,'CALZAGHE')
insert into IT_apps values (600,'LAMOTTA')
insert into IT_apps values (600,'HAGLER')
insert into IT_apps values (600,'HEARNS')
insert into IT_apps values (600,'FRAZIER')
insert into IT_apps values (700,'GUINN')
insert into IT_apps values (700,'JUDAH')
insert into IT_apps values (700,'MARGARITO')
```

```
 1  select max(decode(flag2,0,it_dept)) research,
 2         max(decode(flag2,1,it_dept)) apps
 3    from (
 4  select sum(flag1)over(partition by flag2
 5                            order by flag1,rownum) flag,
 6         it_dept, flag2
 7    from (
 8  select 1 flag1, 0 flag2,
 9         decode(rn,1,to_char(deptno),'  '||ename) it_dept
10    from (
11  select x.*, y.id,
12         row_number()over(partition by x.deptno order by y.id) rn
13    from (
14  select deptno,
15         ename,
16         count(*)over(partition by deptno) cnt
17    from it_research
```

```
18         ) x,
19         (select level id from dual connect by level <= 2) y
20         )
21  where rn <= cnt+1
22 union all
23 select 1 flag1, 1 flag2,
24        decode(rn,1,to_char(deptno),'  '||ename) it_dept
25   from (
26 select x.*, y.id,
27        row_number()over(partition by x.deptno order by y.id) rn
28   from (
29 select deptno,
30        ename,
31        count(*)over(partition by deptno) cnt
32   from it_apps
33        ) x,
34        (select level id from dual connect by level <= 2) y
35        )
36  where rn <= cnt+1
37        ) tmp1
38        ) tmp2
39  group by flag
```

3. 讨论

和其他的数据仓库和报表类型的查询一样，上述解决方案看起来相当复杂，但是如果拆解开来一一细看的话，不难发现它其实是一个 stack-n-pivot 操作，外加一个笛卡儿积（困难重重且没有任何办法）。拆解上述查询的方法是先仔细查看 UNION ALL 的每个组成部分，然后再把它们合并起来做行列变换。先从 UNION ALL 的后半部分开始。

```
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),'  '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
       ) x,
       (select level id from dual connect by level <= 2) y
       ) z
 where rn <= cnt+1

FLAG1      FLAG2 IT_DEPT
----- ---------- -------------------------
    1          1 400
    1          1   MAYWEATHER
    1          1   CASTILLO
    1          1   MARQUEZ
    1          1   MOSLEY
    1          1   CORRALES
    1          1 500
    1          1   CALZAGHE
```

```
1          1   GATTI
1          1 600
1          1   HAGLER
1          1   HEARNS
1          1   FRAZIER
1          1   LAMOTTA
1          1 700
1          1   JUDAH
1          1   MARGARITO
1          1   GUINN
```

仔细看一下上述结果集究竟是怎么拼凑出来的。把上面的查询分解成最基本的组成部分，即可得到内嵌视图 X，该视图从 IT_APPS 表里取出每个 ENAME 和 DEPTNO，并计算出每个 DEPTNO 对应的员工人数。结果如下所示。

```
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps

DEPTNO ENAME                         CNT
------ -------------------- ----------
   400 CORRALES                        5
   400 MAYWEATHER                      5
   400 CASTILLO                        5
   400 MARQUEZ                         5
   400 MOSLEY                          5
   500 GATTI                           2
   500 CALZAGHE                        2
   600 LAMOTTA                         4
   600 HAGLER                          4
   600 HEARNS                          4
   600 FRAZIER                         4
   700 GUINN                           3
   700 JUDAH                           3
   700 MARGARITO                       3
```

下一步是基于内嵌视图 X 返回的行和由 CONNECT BY 从 DUAL 表中产生出来的两行数据创建一个笛卡儿积。该操作的结果集显示如下。

```
select *
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
       ) x,
       (select level id from dual connect by level <= 2) y
 order by 2

DEPTNO ENAME      CNT  ID
------ ---------- --- ---
   500 CALZAGHE     2   1
   500 CALZAGHE     2   2
```

```
   400 CASTILLO     5   1
   400 CASTILLO     5   2
   400 CORRALES     5   1
   400 CORRALES     5   2
   600 FRAZIER      4   1
   600 FRAZIER      4   2
   500 GATTI        2   1
   500 GATTI        2   2
   700 GUINN        3   1
   700 GUINN        3   2
   600 HAGLER       4   1
   600 HAGLER       4   2
   600 HEARNS       4   1
   600 HEARNS       4   2
   700 JUDAH        3   1
   700 JUDAH        3   2
   600 LAMOTTA      4   1
   600 LAMOTTA      4   2
   700 MARGARITO    3   1
   700 MARGARITO    3   2
   400 MARQUEZ      5   1
   400 MARQUEZ      5   2
   400 MAYWEATHER   5   1
   400 MAYWEATHER   5   2
   400 MOSLEY       5   1
   400 MOSLEY       5   2
```

如上所示，由于笛卡儿积的存在内嵌视图 X 的每一行数据都被返回了两次。不用着急，你很快就会明白为什么此处需要一个笛卡儿积。下一步是根据 ID（ID 的值为 1 或 2，这是由笛卡儿积生成的）为当前结果集中每个 DEPTNO 对应的员工进行编号。编号的结果显示在下面查询的输出部分。

```
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
       ) x,
       (select level id from dual connect by level <= 2) y

DEPTNO ENAME      CNT  ID          RN
------ ---------- --- --- ----------
   400 CORRALES     5   1           1
   400 MAYWEATHER   5   1           2
   400 CASTILLO     5   1           3
   400 MARQUEZ      5   1           4
   400 MOSLEY       5   1           5
   400 CORRALES     5   2           6
   400 MOSLEY       5   2           7
   400 MAYWEATHER   5   2           8
   400 CASTILLO     5   2           9
   400 MARQUEZ      5   2          10
```

```
500 GATTI         2    1          1
500 CALZAGHE      2    1          2
500 GATTI         2    2          3
500 CALZAGHE      2    2          4
600 LAMOTTA       4    1          1
600 HAGLER        4    1          2
600 HEARNS        4    1          3
600 FRAZIER       4    1          4
600 LAMOTTA       4    2          5
600 HAGLER        4    2          6
600 FRAZIER       4    2          7
600 HEARNS        4    2          8
700 GUINN         3    1          1
700 JUDAH         3    1          2
700 MARGARITO     3    1          3
700 GUINN         3    2          4
700 JUDAH         3    2          5
700 MARGARITO     3    2          6
```

每个员工都有了一个编号，并且，他们的重复项也都被分配了编号。上述结果集中包含了 IT_APP 表里的所有员工及其重复项，以及基于所属 DEPTNO 为每一行生成的编号。我们需要额外生成这些重复项的原因是，因为我们需要在结果集中留一个缝隙把 DEPTNO 插入到 ENAME 列。如果我们把一个只有 1 行数据的表和 IT_APPS 表连接起来做笛卡儿积，就无法得到这些额外的行。（因为一个表的记录条数乘以 1 的结果仍然会等于该表的记录条数。）

下一步是把到目前为止的结果集做行列翻转操作，这样 ENAMES 会以一列的形式返回，但会先返回他们所属的 DEPTNO。如下所示的查询展示了这一操作过程。

```
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),'  '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
       ) x,
       (select level id from dual connect by level <= 2) y
       ) z
 where rn <= cnt+1

FLAG1      FLAG2 IT_DEPT
----- ---------- ------------------------
    1          1 400
    1          1   MAYWEATHER
    1          1   CASTILLO
    1          1   MARQUEZ
    1          1   MOSLEY
    1          1   CORRALES
    1          1 500
    1          1   CALZAGHE
```

```
     1          1   GATTI
     1          1 600
     1          1   HAGLER
     1          1   HEARNS
     1          1   FRAZIER
     1          1   LAMOTTA
     1          1 700
     1          1   JUDAH
     1          1   MARGARITO
     1          1   GUINN
```

先暂时忽略掉 FLAG1 和 FLAG2，稍后再做讨论。注意观察上述 IT_DEPT 列的结果。每个 DEPTNO 返回的记录行数是 CNT*2，但实际上只需要 CNT+1 行记录，WHERE 子句的过滤条件会限制记录行数。RN 是每个员工的编号，所有编号值小于或等于 CNT+1 的行都会被保留下来。也就是说，每个 DEPTNO 对应的所有员工再加上额外的 1 行记录会被保留下来（额外的那 1 行记录是每个 DEPTNO 对应的编号最小的员工）。这一行额外的记录就是用来插入 DEPTNO 的地方。调用 DECODE 函数（该函数的功能类似于 CASE 表达式，早期 Oracle 版本已经支持该函数）判定 RN 的值，并把 DEPTNO 插入到结果集里。RN 值等于 1 的员工不会被漏掉，但会被放在每个 DEPTNO 的最后位置（因为顺序无关紧要，放在最后也没有关系）。至此为止，我们已经详尽地讨论了 UNION ALL 的后半部分。

UNION ALL 的前半部分过程和后半部分相同，因此就没必要再重复讨论了。下面仔细观察一下两个查询结果叠加后的结果集。

```
select 1 flag1, 0 flag2,
       decode(rn,1,to_char(deptno),'  '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_research
       ) x,
       (select level id from dual connect by level <= 2) y
       )
 where rn <= cnt+1
union all
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),'  '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
       ) x,
       (select level id from dual connect by level <= 2) y
       )
```

```
 where rn <= cnt+1

FLAG1      FLAG2 IT_DEPT
----- ---------- ----------------------
    1          0 100
    1          0   JONES
    1          0   TONEY
    1          0   HOPKINS
    1          0 200
    1          0   P.WHITAKER
    1          0   MARCIANO
    1          0   ROBINSON
    1          0   MORALES
    1          0 300
    1          0   WRIGHT
    1          0   J.TAYLOR
    1          0   LACY
    1          1 400
    1          1   MAYWEATHER
    1          1   CASTILLO
    1          1   MARQUEZ
    1          1   MOSLEY
    1          1   CORRALES
    1          1 500
    1          1   CALZAGHE
    1          1   GATTI
    1          1 600
    1          1   HAGLER
    1          1   HEARNS
    1          1   FRAZIER
    1          1   LAMOTTA
    1          1 700
    1          1   JUDAH
    1          1   MARGARITO
    1          1   GUINN
```

此时你或许还不明白 FLAG1 的作用，但可以看出 FLAG2 被用来标识行记录来自 UNION ALL 的哪个部分（0 表示前半部门，1 表示后半部分）。

下一步是把叠加后的结果集包裹在一个内嵌视图里，并计算 FLAG1 的累计合计值（终于知道它的作用了！），该累计合计值可以看作是 UNION ALL 的两个数据子集内部各自生成的行编号。编号后的结果集（累计合计值）如下所示。

```
select sum(flag1)over(partition by flag2
                          order by flag1,rownum) flag,
       it_dept, flag2
  from (
select 1 flag1, 0 flag2,
       decode(rn,1,to_char(deptno),'  '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno,
```

```
       ename,
       count(*)over(partition by deptno) cnt
  from it_research
       ) x,
       (select level id from dual connect by level <= 2) y
       )
 where rn <= cnt+1
union all
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),'  '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
       ) x,
       (select level id from dual connect by level <= 2) y
       )
where rn <= cnt+1
       ) tmp1

FLAG IT_DEPT              FLAG2
---- --------------- ----------
   1 100                      0
   2   JONES                  0
   3   TONEY                  0
   4   HOPKINS                0
   5 200                      0
   6   P.WHITAKER             0
   7   MARCIANO               0
   8   ROBINSON               0
   9   MORALES                0
  10 300                      0
  11   WRIGHT                 0
  12   J.TAYLOR               0
  13   LACY                   0
   1 400                      1
   2   MAYWEATHER             1
   3   CASTILLO               1
   4   MARQUEZ                1
   5   MOSLEY                 1
   6   CORRALES               1
   7 500                      1
   8   CALZAGHE               1
   9   GATTI                  1
  10 600                      1
  11   HAGLER                 1
  12   HEARNS                 1
  13   FRAZIER                1
  14   LAMOTTA                1
  15 700                      1
  16   JUDAH                  1
```

```
  17   MARGARIT              1
  18   GUINN                 1
```

剩下的最后一步是，基于 FLAG2 把 TMP1 的返回值做行列翻转，同时也要按照 FLAG（TMP1 中生成的累计合计值）进行分组。把 TMP1 的查询结果包裹在一个内嵌视图（最外层的内嵌视图 TMP2）里，并做行列翻转。最终的解决方案和结果集显示如下。

```
select max(decode(flag2,0,it_dept)) research,
       max(decode(flag2,1,it_dept)) apps
  from (
select sum(flag1)over(partition by flag2
                          order by flag1,rownum) flag,
       it_dept, flag2
  from (
select 1 flag1, 0 flag2,
       decode(rn,1,to_char(deptno),'  '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_research
       ) x,
       (select level id from dual connect by level <= 2) y
       )
 where rn <= cnt+1
union all
select 1 flag1, 1 flag2,
       decode(rn,1,to_char(deptno),'  '||ename) it_dept
  from (
select x.*, y.id,
       row_number()over(partition by x.deptno order by y.id) rn
  from (
select deptno deptno,
       ename,
       count(*)over(partition by deptno) cnt
  from it_apps
       ) x,
       (select level id from dual connect by level <= 2) y
       )
 where rn <= cnt+1
       ) tmp1
       ) tmp2
 group by flag

RESEARCH             APPS
-------------------- ---------------
100                  400
  JONES                MAYWEATHER
  TONEY                CASTILLO
  HOPKINS              MARQUEZ
200                    MOSLEY
```

```
  P.WHITAKER          CORRALES
  MARCIANO          500
  ROBINSON            CALZAGHE
  MORALES             GATTI
300                 600
  WRIGHT              HAGLER
  J.TAYLOR            HEARNS
  LACY                FRAZIER
                      LAMOTTA
                    700
                      JUDAH
                      MARGARITO
                      GUINN
```

14.10 在Oracle中把标量子查询转换为复合子查询

1. 问题

一个标量子查询中只允许返回一个值，你想绕过该限制。例如，你尝试执行如下所示的查询。

```
select e.deptno,
       e.ename,
       e.sal,
       (select d.dname,d.loc,sysdate today
          from dept d
         where e.deptno=d.deptno)
  from emp e
```

上述查询会因为报错而无法执行，因为 SELECT 列表里的子查询只允许返回一个值。

2. 解决方案

诚然，上述问题似乎有些不切实际，因为只要把 EMP 表和 DEPT 表连接起来，我们就能方便地从 DEPT 表中提取出任意值。但是，我的本意是希望你关注技巧，并认识到本问题在某些场景下有其实用性。当在 SELECT 中放入了另一个 SELECT（标量子查询）时，要想绕过只返回一个值的限制，就需要利用 Oracle 的对象类型。我们可以定义一个拥有多个属性的对象，然后把它作为一个单独的实体来处理，并且可以访问其中的每一个属性。实际上，我们并没有真正地打破那个规则。它仍然只返回了一个值，只不过该返回值是一个对象，它里面包含了许多属性。

本解决方案使用到了下面的对象类型。

```
create type generic_obj
    as object (
    val1 varchar2(10),
    val2 varchar2(10),
    val3 date
);
```

有了以上对象类型，就可以执行下面的查询。

```
 1  select x.deptno,
 2         x.ename,
 3         x.multival.val1 dname,
 4         x.multival.val2 loc,
 5         x.multival.val3 today
 6    from (
 7  select e.deptno,
 8         e.ename,
 9         e.sal,
10         (select generic_obj(d.dname,d.loc,sysdate+1)
11            from dept d
12           where e.deptno=d.deptno) multival
13    from emp e
14         ) x

DEPTNO ENAME      DNAME      LOC        TODAY
------ ---------- ---------- ---------- -----------
    20 SMITH      RESEARCH   DALLAS     12-SEP-2005
    30 ALLEN      SALES      CHICAGO    12-SEP-2005
    30 WARD       SALES      CHICAGO    12-SEP-2005
    20 JONES      RESEARCH   DALLAS     12-SEP-2005
    30 MARTIN     SALES      CHICAGO    12-SEP-2005
    30 BLAKE      SALES      CHICAGO    12-SEP-2005
    10 CLARK      ACCOUNTING NEW YORK   12-SEP-2005
    20 SCOTT      RESEARCH   DALLAS     12-SEP-2005
    10 KING       ACCOUNTING NEW YORK   12-SEP-2005
    30 TURNER     SALES      CHICAGO    12-SEP-2005
    20 ADAMS      RESEARCH   DALLAS     12-SEP-2005
    30 JAMES      SALES      CHICAGO    12-SEP-2005
    20 FORD       RESEARCH   DALLAS     12-SEP-2005
    10 MILLER     ACCOUNTING NEW YORK   12-SEP-2005
```

3. 讨论

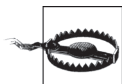
本解决方案的关键在于使用上述对象的构造函数（默认情况下构造函数和对象同名）。因为对象本身是一个标量值，它并不会违反标量子查询的规则，如下所示。

```
select e.deptno,
       e.ename,
       e.sal,
       (select generic_obj(d.dname,d.loc,sysdate-1)
          from dept d
         where e.deptno=d.deptno) multival
  from emp e

DEPTNO ENAME    SAL MULTIVAL(VAL1, VAL2, VAL3)
------ ------ ----- -------------------------------------------------------
    20 SMITH    800 GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2005')
    30 ALLEN   1600 GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2005')
    30 WARD    1250 GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2005')
    20 JONES   2975 GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2005')
    30 MARTIN  1250 GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2005')
    30 BLAKE   2850 GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2005')
```

```
10 CLARK   2450 GENERIC_OBJ('ACCOUNTING', 'NEW YORK', '12-SEP-2005')
20 SCOTT   3000 GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2005')
10 KING    5000 GENERIC_OBJ('ACCOUNTING', 'NEW YORK', '12-SEP-2005')
30 TURNER  1500 GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2005')
20 ADAMS   1100 GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2005')
30 JAMES    950 GENERIC_OBJ('SALES', 'CHICAGO', '12-SEP-2005')
20 FORD    3000 GENERIC_OBJ('RESEARCH', 'DALLAS', '12-SEP-2005')
10 MILLER  1300 GENERIC_OBJ('ACCOUNTING', 'NEW YORK', '12-SEP-2005')
```

接下来，只要把上述查询包裹进一个内嵌视图里，并提取各个属性即可。



重要提示：Oracle 不同于其他数据库，内嵌视图不一定非要有别名。但对于本例而言，我们必须为上述内嵌视图指定一个别名。否则，我们就无法访问对象的属性。

14.11 解析串行化的数据

1. 问题

你有串行化的数据（以字符串形式存储），你希望解析这些字符串并以行的形式返回。例如，你的数据如下所示。

```
STRINGS
-----------------------------------
entry:stewiegriffin:lois:brian:
entry:moe::sizlack:
entry:petergriffin:meg:chris:
entry:willie:
entry:quagmire:mayorwest:cleveland:
entry:::flanders:
Entry:robo:tchi:ken:
```

你希望把这些字符串转换成如下所示的结果集。

```
VAL1            VAL2            VAL3
--------------- --------------- ---------------
moe                             sizlack
petergriffin    meg             chris
quagmire        mayorwest       cleveland
robo            tchi            ken
stewiegriffin   lois            brian
willie
                                flanders
```

2. 解决方案

本例中的每个字符串最多有 3 个值构成。这些值由冒号分隔，但是不一定每个字符串都包含 3 个值。如果一个字符串包含的值不足 3 个，我们必须小心地处理这种状况，确保解析出来的值被放入正确的列里。考虑如下所示的一行数据。

```
entry:::flanders:
```

上面这行数据里缺少了前面两个值，仅剩下第 3 个值。因此，仔细观察“问题”部分给出

的结果集，会发现“flanders”这一行 VAL1 和 VAL2 的值都是 Null。

本解决方案的关键在于遍历字符串并解析字符串，最后再执行一个简单的行列翻转操作。我们还用到了视图 V，下面给出了该视图的定义。另外，这里使用的是 Oracle 语法，由于本实例仅涉及部分字符串解析函数的调用，相信你能很容易地修改代码使之适用于其他数据库。

```
create view V
    as
select 'entry:stewiegriffin:lois:brian:' strings
  from dual
 union all
select 'entry:moe::sizlack:'
  from dual
 union all
select 'entry:petergriffin:meg:chris:'
  from dual
 union all
select 'entry:willie:'
  from dual
 union all
select 'entry:quagmire:mayorwest:cleveland:'
  from dual
 union all
select 'entry:::flanders:'
  from dual
 union all
select 'entry:robo:tchi:ken:'
  from dual
```

视图 V 负责提供示例数据，解决方案如下所示。

```
 1  with cartesian as (
 2  select level id
 3    from dual
 4  connect by level <= 100
 5  )
 6  select max(decode(id,1,substr(strings,p1+1,p2-1))) val1,
 7         max(decode(id,2,substr(strings,p1+1,p2-1))) val2,
 8         max(decode(id,3,substr(strings,p1+1,p2-1))) val3
 9    from (
10  select v.strings,
11         c.id,
12         instr(v.strings,':',1,c.id) p1,
13         instr(v.strings,':',1,c.id+1)-instr(v.strings,':',1,c.id) p2
14    from v, cartesian c
15   where c.id <= (length(v.strings)-length(replace(v.strings,':')))-1
16         )
17   group by strings
18   order by 1
```

3. 讨论

第一步是遍历这些字符串。

```
with cartesian as (
select level id
  from dual
 connect by level <= 100
)
select v.strings,
       c.id
  from v,cartesian c
 where c.id <= (length(v.strings)-length(replace(v.strings,':')))-1

STRINGS                             ID
----------------------------------- ---
entry:::flanders:                     1
entry:::flanders:                     2
entry:::flanders:                     3
entry:moe::sizlack:                   1
entry:moe::sizlack:                   2
entry:moe::sizlack:                   3
entry:petergriffin:meg:chris:         1
entry:petergriffin:meg:chris:         3
entry:petergriffin:meg:chris:         2
entry:quagmire:mayorwest:cleveland:   1
entry:quagmire:mayorwest:cleveland:   3
entry:quagmire:mayorwest:cleveland:   2
entry:robo:tchi:ken:                  1
entry:robo:tchi:ken:                  2
entry:robo:tchi:ken:                  3
entry:stewiegriffin:lois:brian:       1
entry:stewiegriffin:lois:brian:       3
entry:stewiegriffin:lois:brian:       2
entry:willie:                         1
```

下一步是调用 INSTR 函数找出每个字符串里每个冒号的位置。因为我们要提取的每个值都被两个冒号包围，这两个冒号的位置信息分别命名为 P1 和 P2，意思是“位置 1”和“位置 2”[2]。

```
with cartesian as (
select level id
  from dual
 connect by level <= 100
)
select v.strings,
       c.id,
       instr(v.strings,':',1,c.id) p1,
       instr(v.strings,':',1,c.id+1)-instr(v.strings,':',1,c.id) p2
  from v,cartesian c
 where c.id <= (length(v.strings)-length(replace(v.strings,':')))-1
 order by 1

STRINGS                             ID         P1         P2
----------------------------------- --- ---------- ----------
entry:::flanders:                     1          6          1
```

注 2：严格来讲，P2 是两个冒号之间的距离，而不是右侧冒号的索引位置。——译者注

```
entry:::flanders:                    2          7          1
entry:::flanders:                    3          8          9
entry:moe::sizlack:                  1          6          4
entry:moe::sizlack:                  2         10          1
entry:moe::sizlack:                  3         11          8
entry:petergriffin:meg:chris:        1          6         13
entry:petergriffin:meg:chris:        3         23          6
entry:petergriffin:meg:chris:        2         19          4
entry:quagmire:mayorwest:cleveland:  1          6          9
entry:quagmire:mayorwest:cleveland:  3         25         10
entry:quagmire:mayorwest:cleveland:  2         15         10
entry:robo:tchi:ken:                 1          6          5
entry:robo:tchi:ken:                 2         11          5
entry:robo:tchi:ken:                 3         16          4
entry:stewiegriffin:lois:brian:      1          6         14
entry:stewiegriffin:lois:brian:      3         25          6
entry:stewiegriffin:lois:brian:      2         20          5
entry:willie:                        1          6          7
```

现在我们已经知道了每个字符串里每对冒号的位置信息，剩下要做的就是把这些信息传递给 SUBSTR 函数以提取出目标值。我们想要创建一个 3 列的结果集，因此需要调用 DECODE 函数评估上述笛卡儿积里的 ID。

```
with cartesian as (
select level id
  from dual
 connect by level <= 100
)
select decode(id,1,substr(strings,p1+1,p2-1)) val1,
       decode(id,2,substr(strings,p1+1,p2-1)) val2,
       decode(id,3,substr(strings,p1+1,p2-1)) val3
  from (
select v.strings,
       c.id,
       instr(v.strings,':',1,c.id) p1,
       instr(v.strings,':',1,c.id+1)-instr(v.strings,':',1,c.id) p2
  from v,cartesian c
 where c.id <= (length(v.strings)-length(replace(v.strings,':')))-1
       )
 order by 1

VAL1            VAL2            VAL3
--------------- --------------- --------------
moe
petergriffin
quagmire
robo
stewiegriffin
willie

                lois

                meg
                mayorwest
```

```
                tchi
                                brian
                                sizlack
                                chris
                                cleveland
                                flanders
                                ken
```

最后，基于 STRINGS 分组并针对 SUBSTR 的返回值调用聚合函数以生成更有可读性的结果集。

```
with cartesian as (
select level id
  from dual
 connect by level <= 100
)
select max(decode(id,1,substr(strings,p1+1,p2-1))) val1,
       max(decode(id,2,substr(strings,p1+1,p2-1))) val2,
       max(decode(id,3,substr(strings,p1+1,p2-1))) val3
  from (
select v.strings,
       c.id,
       instr(v.strings,':',1,c.id) p1,
       instr(v.strings,':',1,c.id+1)-instr(v.strings,':',1,c.id) p2
  from v,cartesian c
 where c.id <= (length(v.strings)-length(replace(v.strings,':')))-1
       )
 group by strings
 order by 1

VAL1            VAL2            VAL3
--------------- --------------- ---------------
moe                             sizlack
petergriffin    meg             chris
quagmire        mayorwest       cleveland
robo            tchi            ken
stewiegriffin   lois            brian
willie
                                flanders
```

14.12 计算比重

1. 问题

报表中有一组数字值，你想同时显示每个值占总数的百分比。例如，你使用的是 Oracle 数据库，你希望返回一个按照 JOB 维度计算出来的工资分布情况，这样就能判断出哪些 JOB 耗费了公司最多的钱。你也希望把从事每种 JOB 的员工人数包括进来，以避免最终结果集中出现的百分比产生误导。你希望得到如下所示的报表。

```
JOB         NUM_EMPS PCT_OF_ALL_SALARIES
--------- ---------- -------------------
CLERK              4                  14
ANALYST            2                  20
```

```
MANAGER              3                   28
SALESMAN             4                   19
PRESIDENT            1                   17
```

如上所述，如果报表中没有包含员工人数，看起来似乎总经理的工资比例并不高。加入员工人数后，我们才清楚地看到总经理一个人的工资竟然占了总数的 17%。

2. 解决方案

对于本问题而言，只有 Oracle 提供了一个合适的解决方案，因为 Oracle 支持内置函数 RATIO_TO_REPORT。对于其他数据库，为了计算比重，不妨使用除法，请参考 7.11 节。

```
1  select job,num_emps,sum(round(pct)) pct_of_all_salaries
2    from (
3  select job,
4          count(*)over(partition by job) num_emps,
5          ratio_to_report(sal)over()*100 pct
6    from emp
7          )
8   group by job,num_emps
```

3. 讨论

首先使用窗口函数 COUNT OVER 计算每个 JOB 对应的员工人数。然后使用 RATIO_TO_REPORT 计算每个员工的工资占总数的百分比（该值以小数形式返回）。

```
select job,
       count(*)over(partition by job) num_emps,
       ratio_to_report(sal)over()*100 pct
  from emp

JOB         NUM_EMPS        PCT
--------- ---------- ----------
ANALYST            2 10.3359173
ANALYST            2 10.3359173
CLERK              4 2.75624462
CLERK              4 3.78983635
CLERK              4  4.4788975
CLERK              4 3.27304048
MANAGER            3 10.2497847
MANAGER            3 8.44099914
MANAGER            3 9.81912145
PRESIDENT          1 17.2265289
SALESMAN           4 5.51248923
SALESMAN           4 4.30663221
SALESMAN           4 5.16795866
SALESMAN           4 4.30663221
```

最后，使用聚合函数 SUM 计算 RATIO_TO_REPORT 函数的返回值的合计值。不要忘记根据 JOB 和 NUM_EMPS 分组。另外，要乘以 100 才能得到一个代表百分比的整数（例如，对于 25% 来说，应该返回的是 25 而不是 0.25）：

```
select job,num_emps,sum(round(pct)) pct_of_all_salaries
  from (
select job,
```

```
       count(*)over(partition by job) num_emps,
       ratio_to_report(sal)over()*100 pct
  from emp
       )
 group by job,num_emps

JOB         NUM_EMPS PCT_OF_ALL_SALARIES
--------- ---------- -------------------
CLERK              4                  14
ANALYST            2                  20
MANAGER            3                  28
SALESMAN           4                  19
PRESIDENT          1                  17
```

14.13　从Oracle中生成CSV格式的输出

1. 问题

你想把一个表里的数据转换成某种分隔列表形式（例如，以逗号作为分隔符）。例如，对于 EMP 表，你希望得到如下所示的结果集。

```
DEPTNO LIST
------ -------------------------------------
    10 MILLER,KING,CLARK
    20 FORD,ADAMS,SCOTT,JONES,SMITH
    30 JAMES,TURNER,BLAKE,MARTIN,WARD,ALLEN
```

假设你正在使用 Oracle 数据库（Oracle Database 10*g* 或后续的版本），并希望借助 MODEL 子句实现本问题的解决方案。

2. 解决方案

本解决方案利用了 Oracle 的 MODEL 子句提供的迭代功能。这里用到的技巧是，使用窗口函数 ROW_NUMBER OVER 对每个 DEPTNO 对应的员工进行编号（按照 EMPNO 排序，不过按照哪个字段排序并不重要）。因为 MODEL 支持以数组形式访问行值，我们可以通过把序号减 1 实现对前一个数组元素的访问。因此，对于每一行，我们要创建一个列表，该列表包含了当前员工的名字，并加上编号小于当前员工的那些人的名字。

```
 1  select deptno,
 2         list
 3    from (
 4  select *
 5    from (
 6  select deptno,empno,ename,
 7         lag(deptno)over(partition by deptno
 8                             order by empno) prior_deptno
 9    from emp
10         )
11   model
12     dimension by
13     (
14       deptno,
15       row_number()over(partition by deptno order by empno) rn
```

```
16      )
17      measures
18      (
19        ename,
20        prior_deptno,cast(null as varchar2(60)) list,
21        count(*)over(partition by deptno) cnt,
22        row_number()over(partition by deptno order by empno) rnk
23      )
24      rules
25      (
26        list[any,any]
27        order by deptno,rn = case when prior_deptno[cv(),cv()] is null
28                                  then ename[cv( ),cv( )]
29                                  else ename[cv( ),cv( )]||','||
30                                       list[cv(),rnk[cv( ),cv( )]-1]
31                             end
32      )
33      )
34    where cnt = rn
```

3. 讨论

首先使用窗口函数 LAG OVER 读取前一个员工（按照 EMPNO 排序）的 DEPTNO。结果按照 DEPTNO 分区，因此对于每个部门的第一个员工（按照 EMPNO 排序）返回值将是 Null，对于其余员工而言，返回值则是本部门的 DEPTNO。结果集如下所示。

```
select deptno,empno,ename,
       lag(deptno)over(partition by deptno
                           order by empno) prior_deptno
  from emp

DEPTNO      EMPNO ENAME  PRIOR_DEPTNO
------ ---------- ------ ------------
    10       7782 CLARK
    10       7839 KING             10
    10       7934 MILLER           10
    20       7369 SMITH
    20       7566 JONES            20
    20       7788 SCOTT            20
    20       7876 ADAMS            20
    20       7902 FORD             20
    30       7499 ALLEN
    30       7521 WARD             30
    30       7654 MARTIN           30
    30       7698 BLAKE            30
    30       7844 TURNER           30
    30       7900 JAMES            30
```

下一步仔细观察 MODEL 子句的 MEASURES 部分。MEASURES 列表里的项目是如下的几个数组。

- ENAME：一个包含 EMP 表里全部 ENAME 值的数组。
- PRIOR_DEPTNO：由窗口函数 LAG OVER 返回值构成的数组。
- CNT：由每个 DEPTNO 对应的员工人数构成的数组。
- RNK：每个 DEPTNO 分组内每个员工的编号（按照 EMPNO 排序）构成的数组。

数组索引是 DEPTNO 和 RN（在 DIMENSION BY 子句里调用窗口函数 ROW_NUMBER OVER 得到的返回值）。如果希望看到上述这些数组里实际包含了哪些值，只要注释掉 MODEL 子句的 RULES 部分代码并执行查询即可，如下所示。

```
select *
  from (
select deptno,empno,ename,
       lag(deptno)over(partition by deptno
                           order by empno) prior_deptno
  from emp
       )
 model
   dimension by
   (
     deptno,
     row_number()over(partition by deptno order by empno) rn
   )
   measures
   (
     ename,
     prior_deptno,cast(null as varchar2(60)) list,
     count(*)over(partition by deptno) cnt,
     row_number()over(partition by deptno order by empno) rnk
   )
   rules
   (
/*
     list[any,any]
     order by deptno,rn = case when prior_deptno[cv(),cv()] is null
                               then ename[cv(),cv()]
                               else ename[cv(),cv()]||','||
                                    list[cv(),rnk[cv( ),cv( )]-1]
                          end
*/
   )
 order by 1

DEPTNO  RN ENAME  PRIOR_DEPTNO LIST       CNT  RNK
------ --- ------ ------------ ---------- --- ----
    10   1 CLARK                            3    1
    10   2 KING             10              3    2
    10   3 MILLER           10              3    3
    20   1 SMITH                            5    1
    20   2 JONES            20              5    2
    20   4 ADAMS            20              5    4
    20   5 FORD             20              5    5
    20   3 SCOTT            20              5    3
    30   1 ALLEN                            6    1
    30   6 JAMES            30              6    6
    30   4 BLAKE            30              6    4
    30   3 MARTIN           30              6    3
    30   5 TURNER           30              6    5
    30   2 WARD             30              6    2
```

现在我们弄清楚 MODEL 子句里声明的那些项目了，接下来看一下 RULES 部分的代码。CASE 表达式负责评估 PRIOR_DEPTNO 的当前值。如果当前值是 Null，表明它是每个 DEPTNO 分组内的第一个员工，那么该员工的 ENAME 会被作为当前员工的 LIST。如果 PRIOR_DEPTNO 值不是 Null，那么就把前一个员工的 LIST 值附加上当前员工的名字（ENAME 数组），然后把结果作为当前员工的 LIST。针对 DEPTNO 分组内的每一行记录执行该 CASE 表达式操作的话，就会得到一个迭代的、逗号分隔的值列表（即 CSV 格式的输出）。如下所示的例子里打印出了中间结果。

```
select deptno,
       list
  from (
select *
  from (
select deptno,empno,ename,
       lag(deptno)over(partition by deptno
                           order by empno) prior_deptno
  from emp
       )
 model
   dimension by
   (
     deptno,
     row_number()over(partition by deptno order by empno) rn
   )
   measures
   (
     ename,
     prior_deptno,cast(null as varchar2(60)) list,
     count(*)over(partition by deptno) cnt,
     row_number()over(partition by deptno order by empno) rnk
   )
   rules
   (
     list[any,any]
     order by deptno,rn = case when prior_deptno[cv(),cv()] is null
                               then ename[cv(),cv()]
                               else ename[cv(),cv()]||','||
                                    list[cv(),rnk[cv( ),cv( )]-1]
                          end
   )
   )

DEPTNO LIST
------ ---------------------------------------
    10 CLARK
    10 KING,CLARK
    10 MILLER,KING,CLARK
    20 SMITH
    20 JONES,SMITH
    20 SCOTT,JONES,SMITH
    20 ADAMS,SCOTT,JONES,SMITH
    20 FORD,ADAMS,SCOTT,JONES,SMITH
    30 ALLEN
```

```
    30 WARD,ALLEN
    30 MARTIN,WARD,ALLEN
    30 BLAKE,MARTIN,WARD,ALLEN
    30 TURNER,BLAKE,MARTIN,WARD,ALLEN
    30 JAMES,TURNER,BLAKE,MARTIN,WARD,ALLEN
```

最后要过滤掉其他员工，只保留每个 DEPTNO 分组内的最后一个员工，这样才能确保每个 DEPTNO 都能得到一个完整的 CSV 列表。保存在 CNT 和 RN 数组里的值能实现这一点。RN 代表每个 DEPTNO 分组内按照 EMPNO 排序后得到的员工编号，因此每个 DEPTNO 分组内最后一个员工就是满足 CNT = RN 条件的那个员工，如下所示。

```
select deptno,
       list
  from (
select *
  from (
select deptno,empno,ename,
       lag(deptno)over(partition by deptno
                           order by empno) prior_deptno
  from emp
       )
 model
   dimension by
   (
     deptno,
     row_number()over(partition by deptno order by empno) rn
   )
   measures
   (
     ename,
     prior_deptno,cast(null as varchar2(60)) list,
     count(*)over(partition by deptno) cnt,
     row_number()over(partition by deptno order by empno) rnk
   )
   rules
   (
     list[any,any]
     order by deptno,rn = case when prior_deptno[cv(),cv()] is null
                               then ename[cv(),cv()]
                               else ename[cv(),cv()]||','||
                                    list[cv(),rnk[cv( ),cv( )]-1]
                           end
   )
   )
 where cnt = rn

DEPTNO LIST
------ ----------------------------------------
    10 MILLER,KING,CLARK
    20 FORD,ADAMS,SCOTT,JONES,SMITH
    30 JAMES,TURNER,BLAKE,MARTIN,WARD,ALLEN
```

14.14 找出不匹配某个格式的文本

1. 问题

你有一个文本字段，其中包含了一些格式化过的字符串（例如电话号码），你希望找出那些不符合格式要求的值。例如，你的数据如下所示。

```
select emp_id, text
  from employee_comment

EMP_ID     TEXT
---------- ------------------------------------------------------------
7369       126 Varnum, Edmore MI 48829, 989 313-5351
7499       1105 McConnell Court
           Cedar Lake MI 48812
           Home: 989-387-4321
           Cell: (237) 438-3333
```

你希望把电话号码格式不正确的那些行都找出来。举例而言，查询结果应该包括下面这行数据，因为其中包含的那个电话号码同时使用了两种不同的分隔符。

```
7369       126 Varnum, Edmore MI 48829, 989 313-5351
```

你认为一个电话号码里包含的两个分隔符应该使用同样的符号。

2. 解决方案

本问题的解决方案包含多个步骤。

(1) 设法定义何种形式的数据应该被认为“看起来像电话号码”。
(2) 删除格式正确的电话号码。
(3) 查看是否还剩下任何看起来像电话号码的数据。如果是，剩下的那些就是格式不正确的电话号码。

下面的解决方案充分利用了 Oracle Database 10*g* 提供的正则表达式功能。

```
select emp_id, text
  from employee_comment
 where regexp_like(text, '[0-9]{3}[-. ][0-9]{3}[-. ][0-9]{4}')
   and regexp_like(
          regexp_replace(text,
            '[0-9]{3}([-. ])[0-9]{3}\1[0-9]{4}','***'),
          '[0-9]{3}[-. ][0-9]{3}[-. ][0-9]{4}')

    EMP_ID TEXT
---------- ------------------------------------------------------------
      7369 126 Varnum, Edmore MI 48829, 989 313-5351
      7844 989-387.5359
      9999 906-387-1698, 313-535.8886
```

上面的每一行查询结果里都至少包含了一个看起来像电话号码而格式却不符合要求的值。

3. 讨论

本解决方案的关键在于找出那些看起来像电话号码的数据。由于电话号码作为某个文本字段的一部分而存在，该字段包含的任何文本都可能是符合要求的电话号码。我们需要设法缩小搜寻范围，过滤掉那些明显不符合要求的数据。例如，我们不希望在查询结果中看到类似下面的数据。

```
    EMP_ID TEXT
---------- ---------------------------------------------------------
      7900 Cares for 100-year-old aunt during the day. Schedule only
           for evening and night shifts.
```

显然上述数据记录中根本不存在类似电话号码的内容，更别提格式是否有效了。每个人都能看明白这一点。问题是，我们应该如何让关系数据库管理系统也能“明白”这一点。相信你都希望知道具体的做法。请继续读下去吧。



本实例源自 Jonathan Gennick 的一篇文章“*Regular Expression Anti-Patterns*”（已经取得作者授权）。

本解决方案使用下述 Pattern A 定义看起来像电话号码的数据。

```
Pattern A: [0-9]{3}[-. ][0-9]{3}[-. ][0-9]{4}
```

上述 Pattern A 规定了电话号码开头须包含两组 3 位长度的数字，然后须跟着一组 4 位长度的数字。两组之间的分隔符可以是英文破折号（-）、句号（.）或空格。当然，此处我们也能写出一个更复杂的正则表达式。例如，可以考虑允许出现 7 位长度的电话号码。不过，我们没必要偏离主题太远。现在的重点是设法定义何种形式的数据应该被认为“看起来像电话号码”，并且对于本问题而言，Pattern A 已经能够满足要求了。我们也可以写一个不同的正则表达式，不过它和 Pattern A 应该差别不大。

本解决方案的 WHERE 子句使用 Pattern A 确保只有那些可能包含电话号码（必须符合上述正则表达式！）的行才会被筛选出来。

```
select emp_id, text
  from employee_comment
 where regexp_like(text, '[0-9]{3}[-. ][0-9]{3}[-. ][0-9]{4}')
```

接下来，我们需要考虑如何定义一个“格式良好的”电话号码。本解决方案使用 Pattern B 实现该操作。

```
Pattern B: [0-9]{3}([-. ])[0-9]{3}\1[0-9]{4}
```

上述正则表达式里，\1 指代第一个子表达式（[-.]）。必须和 \1 匹配到同样的分隔符。Pattern B 定义了何为格式良好的电话号码，这些电话号码会被排除掉（因为它们的格式符合要求）。本解决方案借助 REGEXP_REPLACE 函数来排除掉这些格式良好的电话号码。

```
regexp_replace(text,
   '[0-9]{3}([-. ])[0-9]{3}\1[0-9]{4}','***'),
```

上述 REGEXP_REPLACE 函数调用出现在 WHERE 子句里。任何格式良好的电话号码都会被三个

连续的星号字符替换掉。同样，Pattern B 也不是一定要写成上述形式，只要它能筛选出符合要求的电话号码即可。

我们已经用三个连续的星号“***”替换掉了格式良好的电话号码，剩下的自然就是那些“看起来像电话号码”、但又不符合格式定义的电话号码了。接着针对 REGEXP_REPLACE 函数的返回值调用 REGEXP_LIKE 函数，这样就能筛选出不符合格式要求的电话号码。

```
and regexp_like(
       regexp_replace(text,
          '[0-9]{3}([-. ])[0-9]{3}\1[0-9]{4}','***'),
       '[0-9]{3}[-. ][0-9]{3}[-. ][0-9]{4}')
```

Oracle 的正则表达式提供了文本模式匹配能力，没有该功能的话，本实例恐怕难以找到合适的解决方案。特别需要指出的是，本实例非常依赖 REGEXP_REPLACE 函数。其他数据库（例如 PostgreSQL）也支持正则表达式。但据我所知，只有 Oracle 同时支持正则表达式匹配和替换功能。

14.15　使用内嵌视图转换数据

1. 问题

你有一个表，其中一列存放的数据有时是数字，有时是字符。具体存放了什么类型的数据记录在同一张表的另外一列里。你希望使用一个子查询把数字类型的数据都提取出来。

```
select *
  from ( select flag, to_number(num) num
       from subtest
       where flag in ('A', 'C') )
       where num > 0
```

不过，上述内嵌视图查询常常返回如下所示的错误信息（但并非总是报错）。

```
ERROR:
ORA-01722: invalid number
```

2. 解决方案

一种办法是强制要求内嵌视图先于外层 SELECT 语句执行。至少在 Oracle 中可以这样做，具体做法是为内层的 SELECT 列表加上伪列 ROWNUM。

```
select *
  from ( select rownum, flag, to_number(num) num
       from subtest
       where flag in ('A', 'C') )
       where num > 0
```

我们会在下面的“讨论”部分具体解释该解决方案的原理。

3. 讨论

之所以出现上述错误是因为有时候优化器会合并内层和外层查询。尽管乍一看去似乎应该先执行内层查询，然后再剔除掉非数字 NUM 值，但实际上真正执行的可能是下面的查询。

```
select flag, to_number(num) num
from subtest
where to_number(num) > 0 and flag in ('A', 'C');
```

现在我们就能清楚地看到出错的原因了：调用 TO_NUMBER 函数之前，含有非数字 NUM 值的行并没有被预先排除掉。



数据库系统应该合并子查询和主查询吗？答案取决于我们考虑问题的角度。应该依据关系理论考虑这个问题，还是把遵从 SQL 标准放在第一位？或者依据某个数据库的具体实现方式来考虑这个问题？

本解决方案向内层查询的 SELECT 列表添加了 ROWNUM，这样一来本问题至少在 Oracle 范围内得到了解决。ROWNUM 是一个函数，它会为每一行查询结果返回一个顺序递增的值。这些顺序递增的值，被称为行号（row number）。如果脱离了具体的查询上下文，Oracle 就无法计算出行号。因此，Oracle 必须先执行子查询并实体化其结果集，这样才能为每一行查询结果计算出正确的行号。也因为这个原因，调用 ROWNUM 函数其实就是一种强迫 Oracle 在执行主查询之前先完整执行子查询的技巧（也就是说，这种状况下 Oracle 不允许合并子查询和主查询）。如果你希望在其他数据库上也这么做，不妨研究一下具体的数据库是否有类似 Oracle 的 ROWNUM 这样的功能。

14.16　测试一组数据中是否存在某个值

1. 问题

你想根据某一组行记录里是否包含某个特定值来生成一个布尔值。试想这样一个例子，一个学生在一段时间内会参加若干场考试。假设他每 3 个月会参加 3 场考试。只要他通过了任何一场，将返回一个标志（flag）以表示考试通过。如果 3 场都没有通过，也会返回一个标志以表示考试未通过。我们看一下下面的例子。（这里使用 Oracle 语法生成所需的示例数据；对于 DB2 和 SQL Server，略微调整即可，因为它们也支持窗口函数。）

```
create view V
as
select 1 student_id,
       1 test_id,
       2 grade_id,
       1 period_id,
       to_date('02/01/2005','MM/DD/YYYY') test_date,
       0 pass_fail
  from dual union all
select 1, 2, 2, 1, to_date('03/01/2005','MM/DD/YYYY'),  1 from dual union all
select 1, 3, 2, 1, to_date('04/01/2005','MM/DD/YYYY'),  0 from dual union all
select 1, 4, 2, 2, to_date('05/01/2005','MM/DD/YYYY'),  0 from dual union all
select 1, 5, 2, 2, to_date('06/01/2005','MM/DD/YYYY'),  0 from dual union all
select 1, 6, 2, 2, to_date('07/01/2005','MM/DD/YYYY'),  0 from dual

select *
  from V
```

```
STUDENT_ID TEST_ID GRADE_ID PERIOD_ID TEST_DATE   PASS_FAIL
---------- ------- -------- --------- ----------- ---------
         1       1        2         1 01-FEB-2005         0
         1       2        2         1 01-MAR-2005         1
         1       3        2         1 01-APR-2005         0
         1       4        2         2 01-MAY-2005         0
         1       5        2         2 01-JUN-2005         0
         1       6        2         2 01-JUL-2005         0
```

仔细观察以上结果集，可以看到这个学生在两个学期里共参加了 6 场考试。他通过了其中一场（1 表示“通过”，0 表示“未通过”），因此他第一个学期的学习成绩算是过关了。他在第二个学期（接下来的 3 个月）没有通过任何一场考试，因此 3 场考试的 PASS_FAIL 列都是 0。你希望返回一个结果集表示这个学生某个学期是否通过了考试。最终你希望得到如下所示的结果集。

```
STUDENT_ID TEST_ID GRADE_ID PERIOD_ID TEST_DATE   METREQ IN_PROGRESS
---------- ------- -------- --------- ----------- ------ -----------
         1       1        2         1 01-FEB-2005      +           0
         1       2        2         1 01-MAR-2005      +           0
         1       3        2         1 01-APR-2005      +           0
         1       4        2         2 01-MAY-2005      -           0
         1       5        2         2 01-JUN-2005      -           0
         1       6        2         2 01-JUL-2005      -           1
```

METREQ（表示是否通过）的值是“+”和“-”，表示学生在一个学期（3 个月）内是否通过了至少一场考试。如果一个学生在一个学期内通过了至少一场考试，则 IN_PROGRESS 值为 0。如果没有通过任何一场考试，那么他参加的最后一场考试对应的 IN_PROGRESS 值应该是 1。

2. 解决方案

本问题比较麻烦的地方是，我们必须把同一组的行看作一个整体，而非互相独立的个体。先看一下“问题”部分给出的 PASS_FAIL 值。如果我们逐行做判断，则除了 TEST_ID 2，其余每行的 METREQ 值都是“-”。但事实并非如此。我们必须基于一组行记录做出判断。借助窗口函数 MAX OVER，很容易确认一个学生在一个学期内是否通过了至少一场考试。一旦掌握了这样的信息，布尔值计算就变成了简单的 CASE 表达式问题。

```
 1 select student_id,
 2        test_id,
 3        grade_id,
 4        period_id,
 5        test_date,
 6        decode( grp_p_f,1,lpad('+',6),lpad('-',6) ) metreq,
 7        decode( grp_p_f,1,0,
 8                decode( test_date,last_test,1,0 ) ) in_progress
 9   from (
10 select V.*,
11        max(pass_fail)over(partition by
12                       student_id,grade_id,period_id) grp_p_f,
13        max(test_date)over(partition by
14                       student_id,grade_id,period_id) last_test
15   from V
16        ) x
```

3. 讨论

本解决方案的关键在于使用窗口函数 MAX OVER 为每个分组返回最大的 PASS_FAIL 值。因为 PASS_FAIL 值只能是 1 或 0，那么只要学生通过了至少一场考试，则 MAX OVER 就会返回 1。下面展示了具体的查询结果。

```
select V.*,
       max(pass_fail)over(partition by
                     student_id,grade_id,period_id) grp_pass_fail
  from V

STUDENT_ID TEST_ID GRADE_ID PERIOD_ID TEST_DATE   PASS_FAIL GRP_PASS_FAIL
---------- ------- -------- --------- ----------- --------- -------------
         1       1        2         1 01-FEB-2005         0             1
         1       2        2         1 01-MAR-2005         1             1
         1       3        2         1 01-APR-2005         0             1
         1       4        2         2 01-MAY-2005         0             0
         1       5        2         2 01-JUN-2005         0             0
         1       6        2         2 01-JUL-2005         0             0
```

以上结果集显示该学生在第一个学期通过了至少一场考试，因此第一个学期对应的 GRP_PASS_FAIL 值都被设置成了 1。除此之外，如果学生在一个学期内没有通过任何一场考试，那么该学期最后一场考试对应的 IN_PROGRESS 值应该被设置为 1。我们也可以使用窗口函数 MAX OVER 来实现这一要求。

```
select V.*,
       max(pass_fail)over(partition by
                     student_id,grade_id,period_id) grp_p_f,
       max(test_date)over(partition by
                     student_id,grade_id,period_id) last_test
  from V

STUDENT_ID TEST_ID GRADE_ID PERIOD_ID TEST_DATE   PASS_FAIL GRP_P_F LAST_TEST
---------- ------- -------- --------- ----------- --------- ------- -----------
         1       1        2         1 01-FEB-2005         0       1 01-APR-2005
         1       2        2         1 01-MAR-2005         1       1 01-APR-2005
         1       3        2         1 01-APR-2005         0       1 01-APR-2005
         1       4        2         2 01-MAY-2005         0       0 01-JUL-2005
         1       5        2         2 01-JUN-2005         0       0 01-JUL-2005
         1       6        2         2 01-JUL-2005         0       0 01-JUL-2005
```

现在，我们已经计算出了这名学生通过了哪个学期的考试以及每个学期最后一场考试的日期，最后只要格式化结果集即可。使用 Oracle 的 DECODE 函数（功能类似 CASE 表达式）计算出最终的 METREQ 和 IN_PROGRESS 列。使用 LPAD 函数确保计算出来的 METREQ 值右对齐。

```
select student_id,
       test_id,
       grade_id,
       period_id,
       test_date,
       decode( grp_p_f,1,lpad('+',6),lpad('-',6) ) metreq,
       decode( grp_p_f,1,0,
               decode( test_date,last_test,1,0 ) ) in_progress
  from (
```

```
select V.*,
       max(pass_fail)over(partition by
                      student_id,grade_id,period_id) grp_p_f,
       max(test_date)over(partition by
                      student_id,grade_id,period_id) last_test
  from V
       ) x

STUDENT_ID TEST_ID GRADE_ID PERIOD_ID TEST_DATE   METREQ IN_PROGRESS
---------- ------- -------- --------- ----------- ------ -----------
         1       1        2         1 01-FEB-2005      +           0
         1       2        2         1 01-MAR-2005      +           0
         1       3        2         1 01-APR-2005      +           0
         1       4        2         2 01-MAY-2005      -           0
         1       5        2         2 01-JUN-2005      -           0
         1       6        2         2 01-JUL-2005      -           1
```

附录A

窗口函数简介

本书的很多实例都使用了 ISO SQL 标准 2003 新增的窗口函数功能，也充分利用了各种数据库的专有窗口函数。本附录将简要介绍窗口函数的工作原理。总体而言，窗口函数的出现使得许多通常被认为非常棘手的任务变得较为容易了（我指的是那些单纯使用标准 SQL 难以解决的问题）。除了阅读本章的内容，你还可以查阅数据库厂商提供的手册，以便获取完整的关于窗口函数列表、语法说明和更深入的工作原理剖析等内容。

A.1 分组

开始讨论窗口函数之前，你需要先了解一下 SQL 的分组是如何工作的。根据我的经验，分组（Grouping）是许多人学习 SQL 的过程中一个不容易跨过去的坎儿。很多时候初学者不了解 `GROUP BY` 子句的工作原理，也不明白为什么使用了 `GROUP BY` 之后查询结果会发生变化。

简单来说，分组就是把相似的行数据聚集在一起。如果一个查询用到了 `GROUP BY`，那么结果集的每一行数据都是一个分组，它们分别代表一行或者几行记录。并且，它们之所以会被分在同一组，是因为这些记录的某一列或者某几列的值相同。以上就是分组的要点。

如果说一个分组就是基于指定的某一列（或者几列）把具有相同值的行数据重新编排而成的一行具有代表性的新数据，那么对于 `EMP` 表而言，显而易见的分组示例包括部门编号等于 10 的全体员工（这些员工之所以被分到同一组，是因为他们的 `DEPTNO` 等于 10），以及全体文员（这些员工的 `JOB='CLERK'`）。我们来看一些查询语句。下列第一个查询给出了部门编号等于 10 的所有员工；第二个查询把部门编号等于 10 的员工分组，并返回一些额外的分组信息：一共有多少行（成员的个数），最高的工资以及最低的工资。

```
select deptno,ename
  from emp
 where deptno=10

DEPTNO ENAME
------ ----------
    10 CLARK
    10 KING
    10 MILLER

select deptno,
       count(*) as cnt,
       max(sal) as hi_sal,
       min(sal) as lo_sal
  from emp
 where deptno=10
 group by deptno

DEPTNO        CNT     HI_SAL     LO_SAL
------ ---------- ---------- ----------
    10          3       5000       1300
```

如果 SQL 不支持把部门编号等于 10 的员工分为一组，那么我们就必须手动检查原始数据才能获得上述第二个查询返回的那些信息。（如果仅有 3 行记录，手动做并不困难；但是，如果有 300 万行记录，又该如何处理呢？）这就引出了另一个问题：我们为什么希望对数据进行分组？有多种可能的原因：或许我们想知道有多少个不同的分组，也或许我们想计算每个分组包含多少个成员（行）。正如上述查询语句所示，借助 SQL 的分组功能，即使一个表里有许多行，我们也能快速获取关于它们的信息，而不必逐行检查原始数据。

A.2 SQL分组的定义

一般而言，数学上的“群”（group）定义为 $(G, \bullet, e)$，其中 G 是一个集合，$\bullet$ 表示 G 的二进制运算，而 e 则是 G 中的成员。我们以该定义为基础来说明什么是 SQL 分组。一个 SQL 分组被定义为 (G, e)，其中 G 是一个带有 `GROUP BY` 子句的、单一的或自足的（self-contained）查询语句的结果集，e 是 G 的成员。并且，一个 SQL 分组须满足下面的两个定理。

- 对于 G 的每一个成员 e，e 具有唯一性，并且存在一个或者多个 e 的实例。
- 对于 G 的每一个成员 e，聚合函数 `COUNT` 的返回值大于 0。



以上 SQL 分组定义里的结果集 G 表明了一个事实，即 SQL 分组的概念依存于 SQL 查询，没有 SQL 查询就不会有 SQL 分组。因此，对于上述定理的表述内容，我们完全可以把 e 替换成“行”，因为 SQL 分组在技术上指的就是由行数据构成的结果集。

上述两个定理是下面讨论 SQL 分组的理论基础，因此有必要先证明它们的正确性（并且在证明的过程当中，我们会像前面一样引入一些具体的 SQL 查询）。

1. 分组不为空

根据上文中的定义，一个分组至少要拥有一个成员（行）。如果我们承认这一点，那么也可以推论出：无法从一个空表生成分组。为证明这个命题的正确性，可以采用反证法，尝试证明它是错误的。下面给出的例子先创建了一个空表，然后尝试通过 3 个不同的查询语句基于该表生成分组。

```
create table fruits (name varchar(10))

select name
  from fruits
 group by name

(no rows selected)


select count(*) as cnt
  from fruits
 group by name

(no rows selected)


select name, count(*) as cnt
  from fruits
 group by name

(no rows selected)
```

上述查询结果说明，我们确实无法从一个空表中生成任何分组。

2. 分组具有唯一性

现在来证明 GROUP BY 子句生成的分组具有唯一性。先向 FRUITS 表插入 5 行记录，然后执行一些查询语句生成分组，如下所示。

```
insert into fruits values ('Oranges')
insert into fruits values ('Oranges')
insert into fruits values ('Oranges')
insert into fruits values ('Apple')
insert into fruits values ('Peach')

select *
  from fruits

NAME
----------
Oranges
Oranges
Oranges
Apple
Peach


select name
```

```
  from fruits
 group by name

NAME
----------
Apple
Oranges
Peach


select name, count(*) as cnt
  from fruits
 group by name

NAME              CNT
---------- ----------
Apple               1
Oranges             3
Peach               1
```

上述第一个查询显示 Oranges 在 FRUITS 表里出现了 3 次。然而，上述第 2 个和第 3 个查询（它们都使用了 GROUP BY）只返回了一行 Oranges。这表明 GROUP BY 结果集里的每一行（即 G 中包含的 e，参见前面的定义）都是唯一的，上述分组的每一个 NAME 值都对应 FRUITS 表的一行或多行记录。

分组具有唯一性，这一点很重要。换句话说，如果查询语句使用了 GROUP BY 子句，那么通常而言 SELECT 列表里就不再需要使用 DISTINCT 关键字了。



我并不是在说 GROUP BY 等价于 DISTINCT。其实它们是两个完全不同的概念。我只是想强调，在结果集里 GROUP BY 子句后面指定的那些字段会先删除重复项，因而没有必要同时使用 DISTINCT 和 GROUP BY。

Frege 的抽象公理和罗素悖论

对于那些有兴趣深入探究的读者，我们不妨介绍一下 Frege 提出的抽象公理，它以 Cantor 的素朴集合论为基础，它指出对于任何一个明确指定的性质，都存在这样一个集合，该集合由所有满足该性质的个体构成。正如 Robert Stoll 曾指出的，该公理的问题在于“对抽象原则的无限制使用”。罗素提醒 Frege 考虑这样一种状况：如果有一个集合，它的成员也是一些集合，并且规定这些集合的性质为“不属于自身”。

正如罗素指出的，Frege 的抽象公理留下了太多自由发挥的空间，如果我们单凭某个条件或性质来定义集合成员关系，那就难免会出现矛盾的状况。为了更形象地解释这个矛盾，罗素提出了“理发师悖论”（Barber Puzzle）。理发师悖论是这样说的：“小镇上有位理发师，他为所有不给自己刮胡子的男人刮胡子，并且只为这些男人刮胡子。可是如此一来，理发师本人的胡子谁来刮呢？”

我们来看一个具体的例子，考虑这样的一个集合：y 的每一个成员 x，都满足指定条件 P，对应的数学符号表示是 $\{x \, \varepsilon \, y \mid P(x)\}$ 。

既然上述集合规定“y 的每一个成员 x，都满足指定条件 P”，那么换一种更直观的说法就是，“当且仅当 x 满足指定条件 P 时，x 是 y 的成员。”

现在我们规定，条件 $P(x)$ 的定义是“x 不是 x 的成员”，即 $(x \sim\varepsilon x)$。

这样一来，上面的表述就变成了“当且仅当 x 不是 x 的成员时，x 是 y 的成员”，即 $\{x\ \varepsilon\ y \mid (x \sim\varepsilon x)\}$ 。

此时，我们不妨问自己一个问题：上述集合是不是它自身的一个成员？在此我们假设 $x = y$，并再次推演一下上述集合的数学表示。如下所示的集合可以被定义为“当且仅当 y 不是 y 的成员时，y 是 y 的成员”，即 $\{y\ \varepsilon\ y \mid (y \sim\varepsilon y)\}$ 。

简而言之，罗素悖论把我们推入了一种两难的境地，有一个集合同时满足两个相互矛盾的条件：它属于其自身，同时又不属于其自身。直觉上我们可能认为这根本就不该成为一个问题。确实，一个集合怎么可能属于自身呢？毕竟，由所有书籍构成的集合并不会是一本书。那么，为什么该悖论会被提出来，并作为一个悖论而存在呢？这是因为当我们站在更高的抽象层面考虑集合理论时，该悖论的存在价值就凸显出来了。例如，罗素悖论的一个相对具有实用价值的运用场景是，探讨“所有集合的集合”问题。对于“所有集合的集合”这样一个概念，根据其定义，它必须是其自身的一个成员（因为它是所有集合的集合）。那么，如果把 $P(x)$ 加于“所有集合的集合”之上，又会如何呢？简单来说，我们参照罗素悖论会推导出“当且仅当所有集合的集合不是其自身的一个成员时，它是其自身的一个成员”，显然这是一个矛盾。

如果你有兴趣追根究底，Ernst Zermelo 后来提出了分离公理模式（也被称作受限概括公理模式或者分类公理），完美地规避了素朴集合论的罗素悖论问题。

3. COUNT永远大于0

前文中的查询语句及其结果也证明了最后一个定理的正确性，即针对一个非空的表执行 GROUP BY 查询，那么聚合函数 COUNT 的返回值不会等于 0。这一点没什么可奇怪的，针对一个分组执行 COUNT 查询确实不可能返回 0。我们已经证明了，无法从一个空表里生成分组，因此一个分组至少会含有 1 行数据。既然至少有 1 行数据，那么 COUNT 查询的结果自然至少等于 1。



注意，以上我们讨论的是同时使用 COUNT 和 GROUP BY 的状况，这和只使用 COUNT 的状况有所不同。如果不要 GROUP BY 子句，针对一个空表执行 COUNT 查询当然会得到 0。

A.3 悖论

“工作结束之际才发现那大厦的基础已经动摇，对于一个科学工作者来说，再也没有比这更为不幸的事情了……在收到罗素先生的一封信之后，我便陷入了两难境地，而此时我的作品即将付印。”

罗素发现了 Frege 的抽象公理中存在的矛盾，上述引言就是 Frege 意识到该问题之后的反应。

悖论通常会针对已有的理论或观点构造出若干矛盾的场景。许多时候，这些矛盾的影响可以被限制在一个较小的范围之内，并且能够通过一些“变通手段”消除其影响。或者，如果它们仅对少数测试案例成立，这样多数时候我们都能放心地忽略掉它们。

你可能猜到了我们为什么要在这里讨论悖论的问题，因为前面给出的 SQL 分组定义确实存在一个悖论，我们必须要在这里认真讨论。尽管到目前为止我们一直在讨论分组的问题，但最终仍然要回到 SQL 查询上。GROUP BY 后面可以接多种语法元素：最常出现的是表的某一列或者几列；除此之外，还可能会出现常量和表达式。我们由此获得了很好的灵活性，但也被迫要付出代价，因为在 SQL 中的 Null 也是一种合法的“值”。Null 的问题在于它会被聚合函数自动忽略掉。比如，如果一个表只有一行一列，并且它的值是 Null，那么如果我们针对该表执行一个既包含 GROUP BY 又包含聚合函数 COUNT 的查询会得到什么结果呢？根据前面给出的定义，如果既有 GROUP BY，又有聚合函数 COUNT，则返回值必然会大于或等于 1。 那么，在函数 COUNT 忽略掉 Null 值的情况下，结果会如何呢？这种状况又会对 SQL 分组定义带来什么样的影响呢？我们来看看下面的例子，它展示了上述 Null 分组悖论。（为增强查询结果的可读性，这里用到了 COALESCE 函数。）

```
select *
  from fruits

NAME
----------
Oranges
Oranges
Oranges
Apple
Peach


insert into fruits values (null)
insert into fruits values (null)
insert into fruits values (null)
insert into fruits values (null)
insert into fruits values (null)

select coalesce(name,'NULL') as name
  from fruits

NAME
----------
Oranges
Oranges
Oranges
Apple
Peach
NULL
NULL
NULL
NULL
NULL
```

```
select coalesce(name,'NULL') as name,
       count(name) as cnt
  from fruits
 group by name

NAME              CNT
---------- ----------
Apple               1
NULL                0
Oranges             3
Peach               1
```

如上所示，正是由于上述 Null 值的出现，我们的 SQL 分组定义现在被迫要面对一个矛盾的场景，即“悖论”。所幸这个矛盾并不是什么大问题，因为它更多地肇始于聚合函数的实现方式，而且和前面给出的定义本身关系不大。对于上述最后面的那个查询语句，如果用自然语言表述问题的话应该是这样的：

> 计算 FRUITS 表里每一种水果出现的次数，即统计每一个分组对应多少个成员。

回头再看一下上面的那些 INSERT 语句，很明显我们一共插入了 5 行 Null 值，也就是说 Null 分组包含 5 个成员。



不同于通常类型的值，Null 有其特殊的性质，它不会等于任何值，但是可以作为分组成员而存在。

查询语句要怎么写才能不返回 0 而返回 5，既能返回我们需要的值，又不违背 SQL 分组的定义呢？如下所示的例子使用了一种变通的做法，消除 Null 分组悖论的影响。

```
select coalesce(name,'NULL') as name,
       count(*) as cnt
  from fruits
 group by name

NAME              CNT
---------- ----------
Apple               1
Oranges             3
Peach               1
NULL                5
```

如果不使用 COUNT(NAME) 而改用 COUNT(*) 的话，就能规避掉 Null 分组悖论。如果传递具体的列名做参数，则聚合函数会忽略掉 Null 值。但是，传递星号 * 而不是列名给 COUNT 函数，返回值就不会等于 0 了。星号 * 迫使 COUNT 函数去计算行数，而不是实际的列值，这样一来不论具体的列值是不是 Null，都不会影响最终的返回值。

另一个悖论针对的是定理“结果集中每一个分组（即 G 中包含的每一个 e）具有唯一性”。从本质上来说，SQL 结果集和表其实不是集合（因为允许出现重复行），更准确的定义应该是多重集合（multiset 或 bag），因此我们有可能生成包含重复分组的结果集。考虑如下所示的查询语句。

```
select coalesce(name,'NULL') as name,
       count(*) as cnt
  from fruits
 group by name
 union all
select coalesce(name,'NULL') as name,
       count(*) as cnt
  from fruits
 group by name

NAME              CNT
---------- ----------
Apple               1
Oranges             3
Peach               1
NULL                5
Apple               1
Oranges             3
Peach               1
NULL                5
```

```
select x.*
  from (
select coalesce(name,'NULL') as name,
       count(*) as cnt
  from fruits
 group by name
       ) x,
       (select deptno from dept) y

NAME              CNT
---------- ----------
Apple               1
Apple               1
Apple               1
Apple               1
Oranges             3
Oranges             3
Oranges             3
Oranges             3
Peach               1
Peach               1
Peach               1
Peach               1
NULL                5
NULL                5
NULL                5
NULL                5
```

上述所示的最终结果中都出现了重复的分组。不过，所幸该悖论仅仅就局部问题展开攻击，因此我们不需要太过担心。分组的第一个性质是，对于 (G,e) 而言，G 是一个带有 GROUP BY 子句的、单一的或自足的查询语句的结果集。简单地说，任何 GROUP BY 查询返回的结果集都符合分组的定义。只有把两个 GROUP BY 查询的结果合并起来的时候，才可能生成含有重复分组的多重集合。上述第一个示例使用了 UNION ALL，这不是一个集合运算，而

是一个多重集合运算，实际上它是两个查询，分两次执行 GROUP BY 操作。



UNION 是一种集合运算，使用 UNION 就不会出现重复分组了。

上述第二个查询使用了笛卡儿积，它首先生成分组，然后再执行笛卡儿积。从一个自足的 GROUP BY 查询的角度来看，它实际上不违背我们的 SQL 分组定义。因此，上述两个例子并没有丝毫撼动我们给出的 SQL 分组定义。相反，它们证明了该定义的完备性，因此我们不会再有所怀疑了。

A.4 SELECT和GROUP BY的关系

前文给出了 SQL 分组的定义并给出了证明，现在可以讨论更多关于 GROUP BY 查询的实际问题。对于 SQL 分组，理解 SELECT 子句和 GROUP BY 子句之间的关系很重要。如果我们要使用诸如 COUNT 这样的聚合函数，就必须记住 SELECT 列表里的任何字段，只要它不是聚合函数的参数，那么它必须成为分组的一部分。比如，对于如下所示的查询语句。

```
select deptno, count(*) as cnt
  from emp
```

DEPTNO 必须被放入 GROUBY 子句。

```
select deptno, count(*) as cnt
  from emp
group by deptno

DEPTNO    CNT
------ ------
    10      3
    20      5
    30      6
```

上述规则并非绝对，也有一些例外情况。例如，常量、用户自定义函数的标量返回值、窗口函数以及非相关标量子查询（non-correlated scalar subquery）。SELECT 子句会在 GROUP BY 子句之后被执行，因而这些语法元素可以出现在 SELECT 列表中，而不必（有些状况下也不允许）出现在 GROUP BY 子句里。例如，

```
select 'hello' as msg,
       1 as num,
       deptno,
       (select count(*) from emp) as total,
       count(*) as cnt
  from emp
group by deptno

MSG   NUM DEPTNO TOTAL CNT
----- --- ------ ----- ---
hello   1     10    14   3
hello   1     20    14   5
hello   1     30    14   6
```

上述查询可能会让你感到困惑。SELECT 列表里那些额外的项目虽然没有被放入 GROUP BY 子句，但是每个 DEPTNO 对应的 CNT 值并不会因此而改变，DEPTNO 的值也不会被改变。那么，我们不妨根据该查询结果修改一下规则，使其表述更加准确。

> 如果 SELECT 列表里的项目可能会改变分组或者改变聚合函数的返回值，则该项目必须被放入 GROUP BY 子句。

上述查询的 SELECT 列表虽然额外添加了一些项目，但是它既不会改变任何分组（每个 DEPTNO）的 CNT 值，也不会改变分组本身。

现在，可以问这样的问题了：到底 SELECT 列表中哪些项目可能会改变分组或者聚合函数的返回值呢？答案很简单：我们想要检索的表的其他列。为上述查询添加一个 JOB 列。

```
select deptno, job, count(*) as cnt
  from emp
 group by deptno, job

DEPTNO JOB       CNT
------ --------- ---
    10 CLERK       1
    10 MANAGER     1
    10 PRESIDENT   1
    20 CLERK       2
    20 ANALYST     2
    20 MANAGER     1
    30 CLERK       1
    30 MANAGER     1
    30 SALESMAN    4
```

为了把额外的 JOB 列也从 EMP 表里提取出来，分组的方式和查询结果集都要随之发生改变。因而我们必须把 DEPTNO 和 JOB 一起放入 GROUP BY 子句，否则上述查询语句不可能正常执行。在 SELECT 和 GROUP BY 子句里都引入 JOB 列，意味着整个查询的语义从“计算每个部门有多少位员工”变成了“计算每个部门有多少个不同的职位”。在这里我要再次提醒你注意，分组具有唯一性；尽管拆开来看的话，DEPTNO 和 JOB 的值有重复，但是每一个 DEPTNO 和 JOB 组合（它们同时出现在 GROUP BY 子句和 SELECT 列表里，因此属于同一个分组）都是独一无二的。（例如，10 和 CLERK 只出现过一次。）

我们也可以在 SELECT 列表里只放聚合函数，这样 GROUP BY 子句里就可以放任何列了。看一下如下所示的两个查询语句，它们展示了这种做法。

```
select count(*)
  from emp
 group by deptno

  COUNT(*)
----------
         3
         5
         6

select count(*)
```

```
  from emp
 group by deptno,job

  COUNT(*)
----------
         1
         1
         1
         2
         2
         1
         1
         1
         4
```

并不是一定要为 SELECT 列表添加一些非聚合函数，但为了提高查询结果的可读性和可用性，我们通常会这么做。



作为一项规定，同时使用 GROUP BY 和聚合函数的时候，对于 SELECT 列表里的任何项目而言，只要它不是聚合函数的参数，那么它就必须被放入 GROUP BY 子句。然而，MySQL 有一个特色功能，它允许我们写出不遵守该规定的 SQL，即我们可以向 SELECT 列表加入一列，该列既不是聚合函数的参数，也不必作为 GROUP BY 子句的一部分而出现。虽然可以宽泛地称之为特色功能，但这其实是一个随时会引发问题的软件缺陷，我强烈建议你不要使用它。事实上，如果你平常使用 MySQL 工作并在意查询语句的精准性，我建议你不使用这个所谓的特色功能。

A.5 窗口操作

一旦理解了分组的概念并掌握了 SQL 聚合运算，就比较容易理解**窗口函数**了。就像聚合函数一样，窗口函数针对指定的行集合（分组）执行聚合运算。不同之处在于，窗口函数能够为每个分组返回多个值，而聚合函数只能返回单一值。聚合运算的对象其实是一组行记录，我们称之为“窗口”（因此才有了术语“窗口函数”）。DB2 把这一类函数称为“OLAP 函数”（在线分析处理函数），Oracle 则称之为“分析函数”（Analytic Function）；本书遵循 ISO SQL 标准的叫法，统一使用术语“窗口函数”。

❑ 一个简单的例子

假设我们希望计算整个公司的员工人数，传统的做法是针对 EMP 表调用 COUNT(*)。

```
select count(*) as cnt
  from emp

   CNT
------
    14
```

以上做法当然非常方便，但是有时候我们可能需要从非聚合数据行或者从不同纬度的聚合

数据行里访问这一类聚合运算结果。窗口函数能帮助我们轻松完成这一类操作。例如，下面的查询语句展示了如何使用窗口函数同时检索出明细行（每个员工一行）和聚合运算结果（员工总人数）。

```
select ename,
       deptno,
       count(*) over() as cnt
  from emp
 order by 2

ENAME      DEPTNO    CNT
---------- ------ ------
CLARK          10     14
KING           10     14
MILLER         10     14
SMITH          20     14
ADAMS          20     14
FORD           20     14
SCOTT          20     14
JONES          20     14
ALLEN          30     14
BLAKE          30     14
MARTIN         30     14
JAMES          30     14
TURNER         30     14
WARD           30     14
```

上述示例调用了窗口函数 `COUNT(*) OVER( )`。关键字 `OVER` 表明 `COUNT` 函数会作为窗口函数来调用，而不是一次普通的聚合函数调用。基本上，SQL 标准中列出的全部聚合函数都能用作窗口函数，关键字 `OVER` 的作用是帮助语法解析器区分不同的使用场景。

那么，`COUNT(*) OVER()` 到底做了什么操作呢？它为上述查询语句返回的每一行数据提供了额外的一列，该列表示 `EMP` 表一共有多少行记录。在这里关键字 `OVER` 后面的圆括号是空的，其实我们也可以在里面放上一个额外的子句，以指明窗口函数操作的行记录范围。保持圆括号里什么也没有，这是明确告知窗口函数把全体行记录作为操作对象，因此上述每一行输出结果都是 14。

我希望你能开始认识到，窗口函数的方便之处在于它可以在一行之中同时执行多种不同维度的聚合运算。后面我们会继续给出更多实例帮助你理解其强大的功能。

A.6　执行时机

在继续深入探讨 `OVER` 子句之前，有必要先理清一个重要的问题，即窗口函数的执行会被安排在整个 SQL 处理的最后一步，但会先于 `ORDER BY` 子句执行。下面举例说明窗口函数的执行时机，这里我们为前一节的查询语句加上一个 `WHERE` 子句，以过滤掉 `DEPTNO` 等于 20 和 30 的员工。

```
select ename,
       deptno,
       count(*) over() as cnt
```

```
  from emp
 where deptno = 10
 order by 2

ENAME      DEPTNO    CNT
---------- ------ ------
CLARK          10      3
KING           10      3
MILLER         10      3
```

每一行的 CNT 值不再是 14，而是变成了 3。在上述示例中，正是因为 WHERE 子句先行把结果集限制为 3 行，才导致窗口函数的返回值变成了 3。（当 SQL 处理到 SELECT 子句的时候，就只剩下 3 行数据留给窗口函数了。）该示例表明 WHERE 和 GROUP BY 这一类子句执行完之后，才轮到窗口函数执行。

A.7　分区

可以使用 PARTITION BY 子句针对行数据进行**分区**（partition）或者分组（group），并根据其结果执行聚合运算。我们在前面的示例中看到过，如果 OVER 关键字后面跟着一个空的圆括号，那么窗口函数执行聚合运算时，会把该查询结果集整体作为一个分区来看待。因此，我们不妨把 PARTITION BY 子句理解成“动态的 GROUP BY”，它不同于传统的 GROUP BY，因为在最终的结果集中允许出现多种由 PARTITION BY 生成的分区。借助 PARTITION BY 针对指定的行数据进行分区和聚合运算（新的分区出现时，聚合运算结果会被重新计算），则所有值（每一个分区的每一个值）都会被返回，而不会只返回一组具有代表性的行记录。下面来看一下如下所示的查询语句。

```
select ename,
       deptno,
       count(*) over(partition by deptno) as cnt
  from emp
 order by 2

ENAME      DEPTNO    CNT
---------- ------ ------
CLARK          10      3
KING           10      3
MILLER         10      3
SMITH          20      5
ADAMS          20      5
FORD           20      5
SCOTT          20      5
JONES          20      5
ALLEN          30      6
BLAKE          30      6
MARTIN         30      6
JAMES          30      6
TURNER         30      6
WARD           30      6
```

上述查询仍然会返回 EMP 表的全部 14 行记录，但是由于使用了 PARTITION BY DEPTNO 子句，

现在聚合函数 COUNT 会分别计算出每一个部门的员工人数。只有当部门发生变化时，聚合运算的结果才会被重新计算，因此同一个部门（同一个分区）的员工会得到相同的 CNT 值。同时我们也要注意到，每一个分区的信息都会被返回，每一个分区的所有成员也都会被返回。和下面的这个查询语句相比，上述使用窗口函数的查询更为高效。

```
select e.ename,
       e.deptno,
       (select count(*) from emp d
         where e.deptno=d.deptno) as cnt
  from emp e
 order by 2

ENAME      DEPTNO    CNT
---------- ------ ------
CLARK          10      3
KING           10      3
MILLER         10      3
SMITH          20      5
ADAMS          20      5
FORD           20      5
SCOTT          20      5
JONES          20      5
ALLEN          30      6
BLAKE          30      6
MARTIN         30      6
JAMES          30      6
TURNER         30      6
WARD           30      6
```

相较于传统的 GROUP BY，PARTITION BY 子句的另一个好处是，在同一个 SELECT 语句里我们可以按照不同的列进行分区，而且不同的窗口函数调用之间互不影响。看一下如下所示的查询，它会逐一列出全体员工，并返回每一个人所属的部门，所在部门的员工总数，每一个人的职位，以及公司范围内从事相同工作的员工总数。

```
select ename,
       deptno,
       count(*) over(partition by deptno) as dept_cnt,
       job,
       count(*) over(partition by job)    as job_cnt
  from emp
 order by 2

ENAME      DEPTNO DEPT_CNT JOB       JOB_CNT
---------- ------ -------- --------- -------
MILLER         10        3 CLERK           4
CLARK          10        3 MANAGER         3
KING           10        3 PRESIDENT       1
SCOTT          20        5 ANALYST         2
FORD           20        5 ANALYST         2
SMITH          20        5 CLERK           4
JONES          20        5 MANAGER         3
ADAMS          20        5 CLERK           4
JAMES          30        6 CLERK           4
```

```
MARTIN          30         6 SALESMAN        4
TURNER          30         6 SALESMAN        4
WARD            30         6 SALESMAN        4
ALLEN           30         6 SALESMAN        4
BLAKE           30         6 MANAGER         3
```

从上述结果集中可以看到，同一个部门的员工会对应相同的 DEPT_CNT 值，从事相同工作的员工也都会得到同样的 JOB_CNT 值。

到目前为止，我们已经讨论完了 PARTITION BY 子句的工作原理，它与 GROUP BY 子句类似，但它不会影响 SELECT 子句的其他项目，也不要求我们写一个 GROUP BY 子句。

A.8 Null的影响

类似于 GROUP BY 子句，PARTITION BY 子句会把所有的 Null 归入同一个分区或者分组。因此，PARTITION BY 对 Null 值的影响也类似于 GROUP BY。下面的查询调用了一个窗口函数来计算每一种业务提成对应的员工人数。（为增强查询结果的可读性，当业务提成为 Null 时返回 –1。）

```
select coalesce(comm,-1) as comm,
       count(*)over(partition by comm) as cnt
  from emp

  COMM        CNT
------ ----------
     0          1
   300          1
   500          1
  1400          1
    -1         10
    -1         10
    -1         10
    -1         10
    -1         10
    -1         10
    -1         10
    -1         10
    -1         10
    -1         10
```

上述查询使用了 COUNT(*)，因而返回值是相应的记录行数。我们看到有 10 个员工的业务提成为 Null。然而，如果不用 *，而改用 COMM 列的话，查询结果就大相径庭了。

```
select coalesce(comm,-1) as comm,
       count(comm)over(partition by comm) as cnt
  from emp

COMM        CNT
---- ----------
   0          1
 300          1
 500          1
```

```
1400        1
  -1        0
  -1        0
  -1        0
  -1        0
  -1        0
  -1        0
  -1        0
  -1        0
  -1        0
  -1        0
```

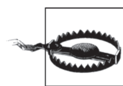
由于上述查询使用了 COUNT(COMM)，它只会计数不为 Null 的 COMM 值。我们看到，业务提成等于 0 的员工有 1 个，等于 300 的有 1 个，等等。但是要特别注意业务提成为 Null 的人数！查询结果是 0，为什么呢？因为聚合函数会忽略掉 NULL 值；更准确地说，聚合函数仅仅计数非 NULL 值。



当使用 COUNT 函数时，我们应该思考一下是否要把 Null 包括在内。使用 COUNT(column) 会忽略 Null。如果希望把 NULL 值一并计入，则应该使用 COUNT(*)。（此时我们要计算的不是实际的列值，而是希望知道有多少行。）

A.9 排序

在使用窗口函数的时候，数据的排序方式可能会对最终的查询结果产生实质性的影响。因此，窗口函数的 OVER 子句也支持 ORDER BY 语法。我们可以使用 ORDER BY 子句指定分区内的行数据如何排序。（记住，如果 OVER 关键字后面没有出现 PARTITION BY 子句，则此处的“分区”指的就是整个查询结果集。）



部分窗口函数强制要求对涉及的分区数据做排序。因此，对于这部分窗口函数而言，ORDER BY 不可省略。在写作本书时，SQL Server 不支持在聚合窗口函数（aggregate window function）的 OVER 子句中使用 ORDER BY。但是，SQL Server 允许在排名窗口函数（window ranking function）的 OVER 子句中使用 ORDER BY。

当在窗口函数的 OVER 子句中使用 ORDER BY 时，我们实际上是在决定两件事。

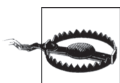
(1) 分区内的行数据如何排序；
(2) 计算涉及哪些行数据。

我们来看一下如下所示的查询，该查询计算出了 DEPTNO 等于 10 的员工的工资累计合计值。

```
select deptno,
       ename,
       hiredate,
       sal,
       sum(sal)over(partition by deptno) as total1,
       sum(sal)over() as total2,
       sum(sal)over(order by hiredate) as running_total
```

```
  from emp
 where deptno=10

DEPTNO ENAME  HIREDATE      SAL TOTAL1 TOTAL2 RUNNING_TOTAL
------ ------ ----------- ----- ------ ------ -------------
    10 CLARK  09-JUN-1981  2450   8750   8750          2450
    10 KING   17-NOV-1981  5000   8750   8750          7450
    10 MILLER 23-JAN-1982  1300   8750   8750          8750
```



注意，其中一个 SUM 查询后面的圆括号是空的。那么，为什么 TOTAL1 和 TOTAL2 查询结果相同呢？这里要再次说明一下，是窗口函数的执行时机决定了上述结果。经过 WHERE 子句的过滤，合计的对象就只剩下 DEPTNO 等于 10 的工资了。在上述示例中只存在一个分区，即整个结果集只包括 DEPTNO 等于 10 的记录。因此，TOTAL1 和 TOTAL2 具有相等的值。

只要看一下 SAL 列的值，就能理解 RUNNING_TOTAL 值是如何计算出来的了。我们还可以逐一累加各个工资值，用实际计算结果验证上述累计合计值的查询结果。但是还有更重要的一点，为什么在计算累计合计值的时候要为 OVER 子句加上 ORDER BY？这是因为在 OVER 子句里加上 ORDER BY 之后，尽管我们看不到，实际上却是在分区内部指定了一个默认的“滑动窗口”。正是由于 ORDER BY HIREDATE 子句的存在才使得合计运算能够自动终止于当前行。

如下所示的查询与前面那个查询等价，它使用了 RANGE BETWEEN 子句显式地指定了 ORDER BY HIREDATE 的默认行为方式。（我们稍后会详细讨论 RANGE BETWEEN 子句。）

```
select deptno,
       ename,
       hiredate,
       sal,
       sum(sal)over(partition by deptno) as total1,
       sum(sal)over() as total2,
       sum(sal)over(order by hiredate
                    range between unbounded preceding
                      and current row) as running_total
  from emp
 where deptno=10

DEPTNO ENAME  HIREDATE      SAL TOTAL1 TOTAL2 RUNNING_TOTAL
------ ------ ----------- ----- ------ ------ -------------
    10 CLARK  09-JUN-1981  2450   8750   8750          2450
    10 KING   17-NOV-1981  5000   8750   8750          7450
    10 MILLER 23-JAN-1982  1300   8750   8750          8750
```

上述查询中出现的 RANGE BETWEEN 子句在 ANSI 标准中被称作 Framing 子句，本书中我将统一使用该术语。现在，你应该已经清楚我们为什么要通过在 OVER 子句中指定 ORDER BY 来计算累计合计值。我们想要告诉查询语句要针对从第一行开始直至当前行为止的全部行记录执行合计运算。（这也是默认行为，ORDER BY 决定“哪些行排在当前行的前面”，就本例而言是按照 HIREDATE 排序。）

A.10 Framing子句

以前面查询的结果集为例，我们来具体解释一下 Framing 子句的工作原理，先从最早入职的员工 CLARK 开始。

(1) 首先 CLARK 的工资是 2450，我们应该把入职时间早于 CLARK 的所有员工的工资一并找出来相加求和。但是，CLARK 是 DEPTNO 等于 10 的部门中最早入职的员工，因此这里只要针对 CLARK 的工资 2450 求和即可，这就是上述查询结果中第一行 RUNNING_TOTAL 值的由来。
(2) 按照 HIREDATE 排序的话，下一个员工是 KING，我们看一下 Framing 子句针对他做了哪些操作。合计运算从当前行的工资 5000（KING 的工资）开始，在此之前的行数据（入职时间早于 KING 的员工的工资）也要包括在内。入职时间比 KING 早的员工只有 CLARK，因而合计值就是 5000 + 2450，计算结果是 7450，这就是上述查询结果中第二行 RUNNING_TOTAL 值的由来。
(3) 按照 HIREDATE 排序的话，分区中最后一个员工是 MILLER，我们再看一下 Framing 子句针对他做了哪些操作。SAL 列的合计运算从当前行的工资 1300（MILLER 的工资）开始，在此之前的行数据（入职时间早于 MILLER 的员工的工资）也要包括在内。CLARK 和 KING 都比 MILLER 入职时间早，因此 MILLER 对应的 RUNNING_TOTAL 要把他们两个人的工资也计入在内：2450+5000+1300，计算结果是 8750，这就是上述查询结果中 MILLER 这一行对应的 RUNNING_TOTAL 值的由来。

如上所述，正是 Framing 子句帮助我们生成了累计合计值。除了规定计算的顺序，ORDER BY 同时也指定了一个默认的滑动窗口。

通常而言，Framing 子句能定义动态变化的“数据子窗口”，并将其融入聚合运算。我们可以使用多种形式的语法指定数据子窗口。考虑如下所示的查询语句。

```
select deptno,
       ename,
       sal,
       sum(sal)over(order by hiredate
                    range between unbounded preceding
                      and current row) as run_total1,
       sum(sal)over(order by hiredate
                     rows between 1 preceding
                      and current row) as run_total2,
       sum(sal)over(order by hiredate
                    range between current row
                      and unbounded following) as run_total3,
       sum(sal)over(order by hiredate
                     rows between current row
                      and 1 following) as run_total4
  from emp
 where deptno=10

DEPTNO ENAME    SAL RUN_TOTAL1 RUN_TOTAL2 RUN_TOTAL3 RUN_TOTAL4
------ ------ ----- ---------- ---------- ---------- ----------
    10 CLARK   2450       2450       2450       8750       7450
    10 KING    5000       7450       7450       6300       6300
    10 MILLER  1300       8750       6300       1300       1300
```

看过上述查询后，你可能会感到难以理解。不过，其实它并不是多么难理解。RUN_TOTAL1就是前面已经解释过的 Framing 子句 UNBOUNDED PRECEDING AND CURRENT ROW。下面我们解释一下其他几个 Framing 子句。

(1) RUN_TOTAL2：不同于关键字 RANGE，此处的 Framing 子句使用了 ROWS，该关键字表明将依据指定数目的行记录产生出滑动窗口。1 PRECEDING 表明起始行是当前行前面的那一行，因而对应的范围就是当前行以及排在它前面的那一行。因此，RUN_TOTAL2 的值就是当前员工的工资加上依照 HIREDATE 排在他前面的那个员工的工资。



非常凑巧的是，此处 CLARK 和 KING 对应的 RUN_TOTAL1 和 RUN_TOTAL2 具有相同值。这是为什么呢？不妨考虑一下，对于这两个员工而言，哪些工资被分别计入了以上两种累计合计值。仔细思考一下，你应该能得到答案。

(2) RUN_TOTAL3：RUN_TOTAL3 与 RUN_TOTAL1 恰好相反；它的计算范围包括当前行以及排在它后面的所有行，而不再是排在它前面的行。

(3) RUN_TOTAL4：和 RUN_TOTAL2 恰好相反，该累计合计值的计算范围包括当前行和排在它后面的一行，而不是排在前面的一行。



你如果能理解到目前为止我们讲述的内容，那么应该也都能领会掌握本书前面给出的各个实例。如果还有不理解的地方，我建议你基于你自己构建的数据和查询语句多加练习和实践。我的个人经验表明，比起仅仅阅读别人写好的查询语句，实际动手写一些代码可能更有助于我们掌握一种新技术。

A.11　最后一个关于Framing子句的例子

最后，我们再来看一个展现 Framing 子句威力的例子。

```
select ename,
       sal,
       min(sal)over(order by sal) min1,
       max(sal)over(order by sal) max1,
       min(sal)over(order by sal
                    range between unbounded preceding
                      and unbounded following) min2,
       max(sal)over(order by sal
                    range between unbounded preceding
                      and unbounded following) max2,
       min(sal)over(order by sal
                    range between current row
                      and current row) min3,
       max(sal)over(order by sal
                    range between current row
                      and current row) max3,
       max(sal)over(order by sal
                     rows between 3 preceding
                      and 3 following) max4
  from emp
```

```
ENAME    SAL  MIN1  MAX1  MIN2  MAX2  MIN3  MAX3  MAX4
------ ----- ------ ------ ------ ------ ------ ------ ------
SMITH    800   800   800   800  5000   800   800  1250
JAMES    950   800   950   800  5000   950   950  1250
ADAMS   1100   800  1100   800  5000  1100  1100  1300
WARD    1250   800  1250   800  5000  1250  1250  1500
MARTIN  1250   800  1250   800  5000  1250  1250  1600
MILLER  1300   800  1300   800  5000  1300  1300  2450
TURNER  1500   800  1500   800  5000  1500  1500  2850
ALLEN   1600   800  1600   800  5000  1600  1600  2975
CLARK   2450   800  2450   800  5000  2450  2450  3000
BLAKE   2850   800  2850   800  5000  2850  2850  3000
JONES   2975   800  2975   800  5000  2975  2975  5000
SCOTT   3000   800  3000   800  5000  3000  3000  5000
FORD    3000   800  3000   800  5000  3000  3000  5000
KING    5000   800  5000   800  5000  5000  5000  5000
```

我们把上述查询结果一一剖开来看。

- MIN1

  生成该字段的窗口函数并没有使用 Framing 子句，因此等价于使用了默认的 Framing 子句 UNBOUNDED PRECEDING AND CURRENT ROW。为什么全部结果行对应的 MIN1 都是 800 呢？这是因为最低工资被排在了最前面（ORDER BY SAL），它始终是最小值。

- MAX1

  MAX1 的值就和 MIN1 大不相同。为什么呢？原因仍然在于默认的 Framing 子句 UNBOUNDED PRECEDING AND CURRENT ROW。由于使用了 ORDER BY SAL，该 Framing 子句的存在使得最高工资始终等于当前行对应的工资。我们来看一下第一行 SMITH。如果比较 SMITH 的工资和排在他前面的工资，我们会发现 SMITH 对应的 MAX1 其实就应该是 SMITH 的工资，因为不存在排在他之前的员工。接着看下一行 JAMES，如果比较 JAMES 的工资和排在他前面的工资，我们发现和 SMITH 的工资相比较的话，JAMES 的工资较高，因此会被筛选出来作为 MAX1 的值。逐一比照每一行数据做一遍推演，最终我们会发现每一行对应的 MAX1 都会等于当前员工的工资。

- MIN2 和 MAX2

  这两列对应的 Framing 子句都是 UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING，实际效果等同于将圆括号置空。函数 MIN 和 MAX 的作用范围是整个结果集。因此，所有行的 MIN2 和 MAX2 值都将是相同的。

- MIN3 和 MAX3

  这两列对应的 Framing 子句都是 CURRENT ROW AND CURRENT ROW，也就是把函数 MIN 和 MAX 的作用范围局限于当前员工的工资。因此，MIN3 和 MAX3 都等于同一行的 SAL 值。这一点应该很容易理解，对吗？

- MAX4

  MAX4 对应的 Framing 子句是 3 PRECEDING AND 3 FOLLOWING，这表明要考虑当前行的前 3 行和后 3 行，当然也包括当前行本身。这样，函数 MAX(SAL) 的调用结果就是对应这些行的最高工资。

以员工 MARTIN 对应的 MAX4 值为例，我们来看一下上述 Framing 子句的处理过程。MARTIN 的工资是 1250，排在他前面的 3 个员工的工资分别是 WARD（1250）、ADAMS（1100）和 JAMES（950）。排在他后面的 3 个员工的工资分别是 MILLER（1300）、TURNER（1500）和 ALLEN（1600）。把 MARTIN 的工资也包括在内，所有这些工资中最高的是 ALLEN 的工资，因此 MARTIN 对应的 MAX4 是 1600。

A.12 代码可读性 + 性能 = 威力

总之，窗口函数非常强大，使用它写出的查询语句能够把明细数据和聚合运算结果融为一体。相比于使用多个自连接和标量子查询，使用窗口函数的代码显得短小精悍。我们来看一下下面的查询语句，它很容易解决如下问题："每个部门有多少名员工？每个部门有多少个不同职位的员工？（例如，DEPTNO 等于 10 的部门有多少个文员。）EMP 表中总共有多少名员工？"

```
select deptno,
       job,
       count(*) over (partition by deptno) as emp_cnt,
       count(job) over (partition by deptno,job) as job_cnt,
       count(*) over () as total
  from emp

DEPTNO JOB          EMP_CNT    JOB_CNT      TOTAL
------ --------- ---------- ---------- ----------
    10 CLERK              3          1         14
    10 MANAGER            3          1         14
    10 PRESIDENT          3          1         14
    20 ANALYST            5          2         14
    20 ANALYST            5          2         14
    20 CLERK              5          2         14
    20 CLERK              5          2         14
    20 MANAGER            5          1         14
    30 CLERK              6          1         14
    30 MANAGER            6          1         14
    30 SALESMAN           6          4         14
    30 SALESMAN           6          4         14
    30 SALESMAN           6          4         14
    30 SALESMAN           6          4         14
```

如果不想使用窗口函数，那就要写一个等价而又稍显麻烦的查询语句。

```
select a.deptno, a.job,
       (select count(*) from emp b
         where b.deptno = a.deptno) as emp_cnt,
       (select count(*) from emp b
         where b.deptno = a.deptno and b.job = a.job) as job_cnt,
       (select count(*) from emp) as total
  from emp a
 order by 1,2
```

```
DEPTNO JOB          EMP_CNT    JOB_CNT      TOTAL
------ --------- ---------- ---------- ----------
    10 CLERK              3          1         14
    10 MANAGER            3          1         14
    10 PRESIDENT          3          1         14
    20 ANALYST            5          2         14
    20 ANALYST            5          2         14
    20 CLERK              5          2         14
    20 CLERK              5          2         14
    20 MANAGER            5          1         14
    30 CLERK              6          1         14
    30 MANAGER            6          1         14
    30 SALESMAN           6          4         14
    30 SALESMAN           6          4         14
    30 SALESMAN           6          4         14
    30 SALESMAN           6          4         14
```

上述不使用窗口函数的方案当然不是很麻烦，但是它的代码显得不够简洁、高效。因为 EMP 表只有 14 行数据，这里无论如何反映不出性能差异，但是如果把数据规模扩大到 1000 行或者 10000 行的话，就应该能看到窗口函数的效率高于多个自连接和标量子查询。

A.13 为报表查询奠定基础

除了在代码可读性和性能方面的优势，窗口函数还为更加复杂的“报表风格”的查询奠定了基础。例如，我们来看一下下面这个“报表风格”的查询，它先在一个内嵌视图中使用了窗口函数，然后在外层查询中展开聚合运算。有了窗口函数，我们能同时得到明细数据和聚合运算结果，这一点对于生成报表非常有帮助。下面的这个查询使用窗口函数计算出了多种不同分区的计数值。由于聚合运算的作用对象是多行数据，内嵌视图将返回 EMP 表中所有行，因此外层的 CASE 表达式能够进一步做形式变换，并生成格式良好的报表。

```
select deptno,
       emp_cnt as dept_total,
       total,
       max(case when job = 'CLERK'
                then job_cnt else 0 end)  as clerks,
       max(case when job = 'MANAGER'
                then job_cnt else 0  end) as mgrs,
       max(case when job = 'PRESIDENT'
                then job_cnt else 0  end) as prez,
       max(case when job = 'ANALYST'
                then job_cnt else 0  end) as anals,
       max(case when job = 'SALESMAN'
                then job_cnt else 0  end) as smen
  from (
select deptno,
       job,
       count(*) over (partition by deptno) as emp_cnt,
       count(job) over (partition by deptno,job) as job_cnt,
       count(*) over () as total
  from emp
       ) x
```

```
 group by deptno, emp_cnt, total

DEPTNO DEPT_TOTAL TOTAL CLERKS MGRS PREZ ANALS SMEN
------ ---------- ----- ------ ---- ---- ----- ----
    10          3    14      1    1    1     0    0
    20          5    14      2    1    0     2    0
    30          6    14      1    1    0     0    4
```

上述查询返回了所有部门，每个部门中员工总人数，EMP 表中员工总人数，以及每个部门中每一种职位各有多少个员工。更值得注意的是，只要一个查询就能完成所有这些工作，而且不需要用到任何连接查询或者临时表。

下面给出最后一个实例，它再次展示了使用窗口函数能够多么方便地同时给出多个问题的答案。

```
select ename as name,
       sal,
       max(sal)over(partition by deptno) as hiDpt,
       min(sal)over(partition by deptno) as loDpt,
       max(sal)over(partition by job) as hiJob,
       min(sal)over(partition by job) as loJob,
       max(sal)over() as hi,
       min(sal)over() as lo,
       sum(sal)over(partition by deptno
                        order by sal,empno) as dptRT,
       sum(sal)over(partition by deptno) as dptSum,
       sum(sal)over() as ttl
  from emp
 order by deptno,dptRT


NAME     SAL HIDPT LODPT HIJOB LOJOB    HI   LO  DPTRT DPTSUM    TTL
------ ----- ----- ----- ----- ----- ----- ---- ------ ------ ------
MILLER  1300  5000  1300  1300   800  5000  800   1300   8750  29025
CLARK   2450  5000  1300  2975  2450  5000  800   3750   8750  29025
KING    5000  5000  1300  5000  5000  5000  800   8750   8750  29025
SMITH    800  3000   800  1300   800  5000  800    800  10875  29025
ADAMS   1100  3000   800  1300   800  5000  800   1900  10875  29025
JONES   2975  3000   800  2975  2450  5000  800   4875  10875  29025
SCOTT   3000  3000   800  3000  3000  5000  800   7875  10875  29025
FORD    3000  3000   800  3000  3000  5000  800  10875  10875  29025
JAMES    950  2850   950  1300   800  5000  800    950   9400  29025
WARD    1250  2850   950  1600  1250  5000  800   2200   9400  29025
MARTIN  1250  2850   950  1600  1250  5000  800   3450   9400  29025
TURNER  1500  2850   950  1600  1250  5000  800   4950   9400  29025
ALLEN   1600  2850   950  1600  1250  5000  800   6550   9400  29025
BLAKE   2850  2850   950  2975  2450  5000  800   9400   9400  29025
```

上述查询简单、高效地回答了下列几个问题，并且保持了很好的代码可读性（甚至没有使用额外的连接查询）。它只需要把员工及其工资和结果集中的相对应的行数据匹配起来即可。

(1) 在所有员工中谁的工资最高（HI）；

(2) 在所有员工中谁的工资最低（LO）；
(3) 在每个部门里谁的工资最高（HIDPT）；
(4) 在每个部门里谁的工资最低（LODPT）；
(5) 在每个工作种类里谁的工资最高（HIJOB）；
(6) 在每个工作种类里谁的工资最低（LOJOB）；
(7) 全部工资的合计值（TTL）；
(8) 每个部门工资的合计值（DPTSUM）；
(9) 每个部门的工资累计合计值（DPTRT）。

附 录 B

重温Rozenshtein

附录 B 是为了向 David Rozenshtein 致敬。我在前言部分说过，我认为他的作品 *The Essence of SQL* 即使到了今天仍然是最棒的 SQL 著作。这本只有 119 页的作品囊括了我心目中一个 SQL 程序员应该掌握的最重要的知识。更为可贵的是，David Rozenshtein 在书中向我们展示了如何深入思考一个问题，并最终找到解决办法。书中提供的解决方案具有典型的面向集合思考的特征。或许在实际环境中由于表的体积太大，我们无法照搬书中提供的解决方案，但它提供的思路非常有启发性，能够引导我们基于集合的思考方式去探索解决方案，而不是陷入过程化的迷思。

The Essence of SQL 一书的出版时间远在窗口函数和 `MODEL` 子句出现之前。本附录会尽量利用标准 SQL 新近提供的功能为这本书中提出的一些问题找出新的解决方案。（至于这些解决方案是否优于旧方案，需要根据实际情况而定。）在每一个讨论部分的最后，我都会列出 *The Essence of SQL* 中提供的解决方案。部分示例的问题可能会在 *The Essence of SQL* 的基础上做一些改动，在这种状况下，我也会适当地修改原有解决方案（改动后，仍然会使用和原解决方案相似的做法和技巧）。

B.1　示例表和数据

下面的表都来自 *The Essence of SQL* 一书，本附录后面的实例会用到它们。

```
/* student表*/
create table student
( sno    integer,
  sname  varchar(10),
  age    integer
)
```

```
/* courses表*/
create table courses
( cno      varchar(5),
  title    varchar(10),
  credits  integer
)

/* professor表 */
create table professor
( lname   varchar(10),
  dept    varchar(10),
  salary  integer,
  age     integer
)

/* student表和学生选修的课程*/
create table take
( sno    integer,
  cno    varchar(5)
)


/* professor表和教授所讲的课程*/
create table teach
( lname   varchar(10),
  cno     varchar(5)
)

insert into student values (1,'AARON',20)
insert into student values (2,'CHUCK',21)
insert into student values (3,'DOUG',20)
insert into student values (4,'MAGGIE',19)
insert into student values (5,'STEVE',22)
insert into student values (6,'JING',18)
insert into student values (7,'BRIAN',21)
insert into student values (8,'KAY',20)
insert into student values (9,'GILLIAN',20)
insert into student values (10,'CHAD',21)

insert into courses values ('CS112','PHYSICS',4)
insert into courses values ('CS113','CALCULUS',4)
insert into courses values ('CS114','HISTORY',4)

insert into professor values ('CHOI','SCIENCE',400,45)
insert into professor values ('GUNN','HISTORY',300,60)
insert into professor values ('MAYER','MATH',400,55)
insert into professor values ('POMEL','SCIENCE',500,65)
insert into professor values ('FEUER','MATH',400,40)

insert into take values (1,'CS112')
insert into take values (1,'CS113')
insert into take values (1,'CS114')
insert into take values (2,'CS112')
insert into take values (3,'CS112')
insert into take values (3,'CS114')
```

```
insert into take values (4,'CS112')
insert into take values (4,'CS113')
insert into take values (5,'CS113')
insert into take values (6,'CS113')
insert into take values (6,'CS114')

insert into teach values ('CHOI','CS112')
insert into teach values ('CHOI','CS113')
insert into teach values ('CHOI','CS114')
insert into teach values ('POMEL','CS113')
insert into teach values ('MAYER','CS112')
insert into teach values ('MAYER','CS114')
```

B.2　逻辑否定问题

David Rozenshtein 在 *The Essence of SQL* 一书中提出了各种类型的基础问题，以此来考察你对 SQL 的掌握程度。你在实际工作中通常也会遇到以这样或那样的形式出现的这类问题。逻辑否定就是其中一种。我们经常需要找出不符合某些特定条件的行。如果查询条件简单，自然很容易解决。但正如下面的例子所展现的那样，有一些逻辑否定问题确实要求我们加入一点创造性的思考才能解决。

B.2.1　问题1

你希望找出没有选修过 CS112 课程的学生，但下面的查询语句返回的结果却是错误的。

```
select *
  from student
 where sno in ( select sno
                  from take
                 where cno != 'CS112' )
```

一个学生可能会选修多门课程，而以上查询却有可能把选修了 CS112 的学生也提取出来。该查询之所以不正确，是因为它没有正确回答问题："谁没有选修 CS112？"实际上，它回答的问题是："谁选修了 CS112 之外的课程？"正确的结果集应该包括没有选修任何课程的学生，以及选修了一些课程却没有选 CS112 的学生。最终，你希望得到如下所示的结果集。

```
      SNO SNAME             AGE
--------- ---------- ----------
        5 STEVE              22
        6 JING               18
        7 BRIAN              21
        8 KAY                20
        9 GILLIAN            20
       10 CHAD               21
```

MySQL 和 PostgreSQL

使用 CASE 表达式和聚合函数 MAX 标识一个学生是否选修了 CS112 课程。

```
1 select s.sno,s.sname,s.age
2   from student s left join take t
3     on (s.sno = t.sno)
```

```
4   group by s.sno,s.sname,s.age
5  having max(case when t.cno = 'CS112'
6                  then 1 else 0 end) = 0
```

DB2 和 SQL Server

使用 CASE 表达式和窗口函数 MAX OVER 标识一个学生是否选修了 CS112 课程。

```
 1  select distinct sno,sname,age
 2    from (
 3  select s.sno,s.sname,s.age,
 4         max(case when t.cno = 'CS112'
 5                  then 1 else 0 end)
 7         over(partition by s.sno,s.sname,s.age) as takes_CS112
 9    from student s, take t
10      on (s.sno = t.sno)
11         ) x
12   where takes_CS112 = 0
```

Oracle

对于 Oracle 9*i* 及其后续版本，上述 DB2 解决方案也适用。除此之外，还可以使用 Oracle 专有的外连接语法，Oracle 8*i* 或更早版本则只能使用该语法。

```
/* group by解决方案*/

1  select s.sno,s.sname,s.age
2    from student s, take t
3   where s.sno = t.sno (+)
4   group by s.sno,s.sname,s.age
5  having max(case when t.cno = 'CS112'
6                  then 1 else 0 end) = 0

/* 窗口函数解决方案 */

 1  select distinct sno,sname,age
 2    from (
 3  select s.sno,s.sname,s.age,
 4         max(case when t.cno = 'CS112'
 5                  then 1 else 0 end)
 7         over(partition by s.sno,s.sname,s.age) as takes_CS112
 9    from student s, take t
10   where s.sno = t.sno (+)
11         ) x
12   where takes_CS112 = 0
```

讨论

以上几种解决方案的语法各有不同，但其做法并无二致。这里用到的技巧是，在结果集中创建一个“布尔”列来表示学生是否选修了 CS112 课程。如果一个学生选修了 CS112，那么该列的值为 1；否则，为 0。下面的查询把 CASE 表达式移到了 SELECT 列表里，并把中间结果打印出来。

```
select s.sno,s.sname,s.age,
       case when t.cno = 'CS112'
            then 1
```

```
              else 0
         end as takes_CS112
    from student s left join take t
      on (s.sno=t.sno)

SNO SNAME             AGE TAKES_CS112
--- ---------- ---------- -----------
  1 AARON              20           1
  1 AARON              20           0
  1 AARON              20           0
  2 CHUCK              21           1
  3 DOUG               20           1
  3 DOUG               20           0
  4 MAGGIE             19           1
  4 MAGGIE             19           0
  5 STEVE              22           0
  6 JING               18           0
  6 JING               18           0
  8 KAY                20           0
 10 CHAD               21           0
  7 BRIAN              21           0
  9 GILLIAN            20           0
```

外连接到 TAKE 表是为了确保把那些没有选修任何课程的学生也能被筛选出来。接着要调用 MAX 函数找出最大的 CASE 表达式返回值。如果一个学生选修了 CS112 课程，最大值会是 1，因为其他课程对应的值都是 0。对于 GROUP BY 解决方案而言，最后一步借助 HAVING 子句筛选出 MAX/CASE 表达式返回值等于 0 的学生。对于窗口函数解决方案，我们需要把上面的查询放入一个内嵌视图，并在外层查询中引用 TAKES_CS112，因为 WHERE 子句不能引用窗口函数的查询结果。另外，根据窗口函数的工作原理，这里也必须剔除掉重复的课程。

原解决方案

The Essence of SQL 一书中对于本问题的解决方案非常巧妙，如下所示。

```
select *
  from student
 where sno not in (select sno
                     from take
                    where cno = 'CS112')
```

我们可以这么理解其思路：“在 TAKE 表中找出选修 CS112 课程的学生，然后从 STUDENT 表中找出不在其中的学生。”该做法遵从了 David Rozenshtein 在该书的最后部分给出的有关逻辑否定的建议：“要记住真正的逻辑否定要求两个步骤，即为了找出‘哪些人不是’，就要先找出‘哪些人是’，然后再排除掉他们。”

B.2.2　问题2

你希望找出只选修了 CS112 和 CS114 中的一门，而不是两门都选的学生。下面的查询语句看似有道理，但返回的结果集却不对。

```
select *
  from student
```

```
where sno in ( select sno
                 from take
                where cno != 'CS112'
                  and cno != 'CS114' )
```

选修了至少一门课程的学生里面，只有 DOUG 和 AARON 同时选了 CS112 和 CS114。这两个学生应该被排除在外。学生 STEVE 选修了 CS113，但既不是 CS112，也不是 CS114，所以也应该被排除在外。

因为一个学生会选修多门课程，在这里我们应该为每个学生只返回一行记录，并创建一个字段用于标记该学生是否选了 CS112 或 CS114，或者同时选了这两门课程。这种做法使得我们能够很容易确认一个学生是否同时选修了这两门课程，并且不必多次扫描原表数据。最终的结果集如下所示。

```
SNO SNAME             AGE
--- ---------- ----------
  2 CHUCK              21
  4 MAGGIE             19
  6 JING               18
```

MySQL 和 PostgreSQL

使用 CASE 表达式和聚合函数 SUM 找出选修 CS112 或 CS114，但又没有同时选这两门课程的学生。

```
1  select s.sno,s.sname,s.age
2    from student s, take t
3   where s.sno = t.sno
4   group by s.sno,s.sname,s.age
5  having sum(case when t.cno in ('CS112','CS114')
6                  then 1 else 0 end) = 1
```

DB2、Oracle 和 SQL Server

使用 CASE 表达式和窗口函数 SUM OVER 找出选修了 CS112 或 CS114，但又没有同时选这两门课程的学生。

```
1  select distinct sno,sname,age
2    from (
3  select s.sno,s.sname,s.age,
4         sum(case when t.cno in ('CS112','CS114') then 1 else 0 end)
5         over (partition by s.sno,s.sname,s.age) as takes_either_or
6    from student s, take t
7   where s.sno = t.sno
8         )x
9   where takes_either_or = 1
```

讨论

解决本问题的第一步是，内连接 STUDENT 表和 TAKE 表，这样就排除了那些没有选修任何课程的学生。接着使用 CASE 表达式标记一个学生是否选修了这两门课程中的一门。在下面的查询语句里，我们把 CASE 表达式移入到 SELECT 列表里，并且打印出中间结果。

```
select s.sno,s.sname,s.age,
       case when t.cno in ('CS112','CS114')
```

```
            then 1 else 0 end as takes_either_or
  from student s, take t
 where s.sno = t.sno

SNO SNAME      AGE TAKES_EITHER_OR
--- ---------- --- ---------------
  1 AARON       20               1
  1 AARON       20               0
  1 AARON       20               1
  2 CHUCK       21               1
  3 DOUG        20               1
  3 DOUG        20               1
  4 MAGGIE      19               1
  4 MAGGIE      19               0
  5 STEVE       22               0
  6 JING        18               0
  6 JING        18               1
```

TAKES_EITHER_OR 值等于 1，表示该学生选修了 CS112 或者 CS114。因为一个学生可以选修多门课程，下一步要使用 GROUP BY 和聚合函数 SUM 为每个学生返回一行记录。函数 SUM 会把每个学生对应的 1 都累加起来。

```
select s.sno,s.sname,s.age,
       sum(case when t.cno in ('CS112','CS114')
                then 1 else 0 end) as takes_either_or
  from student s, take t
 where s.sno = t.sno
 group by s.sno,s.sname,s.age

SNO SNAME      AGE TAKES_EITHER_OR
--- ---------- --- ---------------
  1 AARON       20               2
  2 CHUCK       21               1
  3 DOUG        20               2
  4 MAGGIE      19               1
  5 STEVE       22               0
  6 JING        18               1
```

既没有选修 CS112 也没有选 CS114 的学生对应的 TAKES_EITHER_OR 值是 0。同时选了 CS112 和 CS114 的学生对应 TAKES_EITHER_OR 值是 2。我们希望保留 TAKES_EITHER_OR 值等于 1 的学生。最终的解决方案使用 HAVING 子句筛选出 TAKES_EITHER_OR 值等于 1 的学生。

对于窗口函数解决方案，具体做法也类似。这里同样需要把上述查询放入内嵌视图里，然后在外层查询中引用 TAKES_EITHER_OR 列，因为不能直接在 WHERE 子句中引用窗口函数。（在 SQL 处理过程中，窗口函数是最后被评估执行的部分，但会先于 ORDER BY 子句执行。）另外，根据窗口函数的工作原理，这里也必须剔除掉重复的课程。

原解决方案

下面的查询语句是 *The Essence of SQL* 一书中的解决方案（略有改动）。该查询和问题 1 中的原解决方案的做法相同，都非常巧妙。该解决方案使用自连接找出同时选修了 CS112 和 CS114 的学生，然后使用子查询从选修了 CS112 或 CS114 的学生中把同时选了两门的学生剔除掉。

```
select *
  from student s, take t
 where s.sno = t.sno
   and t.cno in ( 'CS112', 'CS114' )
   and s.sno not in ( select a.sno
                        from take a, take b
                       where a.sno = b.sno
                         and a.cno = 'CS112'
                         and b.cno = 'CS114' )
```

B.2.3 问题3

你希望找出选修了 CS112，而且没有选修其他课程的学生，但下面的查询语句返回的结果不正确。

```
select s.*
  from student s, take t
 where s.sno = t.sno
   and t.cno = 'CS112'
```

CHUCK 是唯一选修了 CS112，而且没有选修其他课程的学生，上述查询应该只返回 CHUCK。

本问题可以被理解为"找出只选修了 CS112 的学生"。上述查询筛选出了选修 CS112 的学生，但同时选了其他课程的学生也会被一并返回。正确的查询语句回答的问题应该是"谁只选修了一门课程，并且这门课是 CS112"。

MySQL 和 PostgreSQL

使用聚合函数 COUNT 确保下列查询返回的学生只选修了一门课程。

```
 1 select s.*
 2   from student s,
 3        take t1,
 4        (
 5 select sno
 6   from take
 7  group by sno
 8 having count(*) = 1
 9        ) t2
10  where s.sno  = t1.sno
11    and t1.sno = t2.sno
12    and t1.cno = 'CS112'
```

DB2、Oracle 和 SQL Server

使用窗口函数 COUNT OVER 确保下列查询返回的学生只选修了一门课程。

```
1  select sno,sname,age
2    from (
3  select s.sno,s.sname,s.age,t.cno,
4         count(t.cno) over (
5           partition by s.sno,s.sname,s.age
6         ) as cnt
7    from student s, take t
8   where s.sno = t.sno
```

```
 9         ) x
10   where cnt = 1
11     and cno = 'CS112'
```

讨论

本解决方案的关键要写一个查询来回答这两个问题："哪些学生只选修了一门课程？""哪些学生选修了 CS112 课程？"第一种解决方案使用内嵌视图 T2 找出只选修了一门课程的学生。接着连接内嵌视图 T2 到 TAKE 表，并且筛选出选修 CS112 课程的学生。（这样一来，剩下的就是只选修一门课程并且那门课程是 CS112 的学生。）下面的查询给出了到目前为止的结果。

```
select t1.*
  from take t1,
       (
select sno
  from take
 group by sno
having count(*) = 1
       ) t2
 where t1.sno = t2.sno
   and t1.cno = 'CS112'

SNO CNO
--- -----
  2 CS112
```

最后，在内嵌视图 T2 和 TAKE 表连接查询的基础上再次连接 STUDENT 表，找出匹配的学生。上述窗口函数解决方案也采取了类似的做法，但是处理方式上稍有不同（更有效率）。内嵌视图 X 返回了每一个学生、他们选修的课程以及他们选修了几门课程。（TAKE 表和 STUDENT 表之间的内连接确保没有选修任何课程的学生会被剔除掉。）结果集如下所示。

```
select s.sno,s.sname,s.age,t.cno,
       count(t.cno) over (
         partition by s.sno,s.sname,s.age
       ) as cnt
  from student s, take t
 where s.sno = t.sno

SNO SNAME             AGE CNO          CNT
--- ---------- ---------- ----- ----------
  1 AARON              20 CS112          3
  1 AARON              20 CS113          3
  1 AARON              20 CS114          3
  2 CHUCK              21 CS112          1
  3 DOUG               20 CS112          2
  3 DOUG               20 CS114          2
  4 MAGGIE             19 CS112          2
  4 MAGGIE             19 CS113          2
  5 STEVE              22 CS113          1
  6 JING               18 CS113          2
  6 JING               18 CS114          2
```

获得了每个学生选修的课程和课程数目之后，最后只要保留 CNT 等于 1 并且 CNO 等于 CS112 的行即可。

原解决方案

The Essence of SQL 一书中的解决方案使用了子查询和双重否定。

```
select s.*
  from student s, take t
 where s.sno = t.sno
   and s.sno not in ( select sno
                        from take
                       where cno != 'CS112' )
```

这是一个非常巧妙的解决方案，上述查询既没有计算每个学生选修了几门课程，也没有设置过滤条件确保查询返回的学生选修了 CS112。那么，它是怎么做到的呢？其中的子查询负责找出至少选修了一门课，但又没有选修 CS112 的所有学生，结果集显示如下。

```
select sno
  from take
 where cno != 'CS112'

SNO
----
   1
   1
   3
   4
   5
   6
   6
```

外层查询负责找出选修了一门课程（任意课程），并且不在上述子查询的返回结果的学生。先忽略外层查询里的 NOT IN 部分，我们能看到如下所示的中间结果集（打印出了所有至少选修了一门课程的学生）。

```
select s.*
  from student s, take t
 where s.sno = t.sno

SNO SNAME             AGE
--- ---------- ----------
  1 AARON              20
  1 AARON              20
  1 AARON              20
  2 CHUCK              21
  3 DOUG               20
  3 DOUG               20
  4 MAGGIE             19
  4 MAGGIE             19
  5 STEVE              22
  6 JING               18
  6 JING               18
```

如果仔细比较上述两个结果集，我们就会发现外层查询的 NOT IN 实际上执行了差集运算，用外层查询得到的 SNO 减去子查询的 SNO，结果只剩下 SNO 等于 2 的学生。总之，子查询负责找出没有选修 CS112 课程的学生。外层查询负责找出不属于上述子查询返回结果集的学生。（这样剩下的学生就是只选修 CS112 课程的学生且没有选修任何课程的学生。）STUDENT 表和 TAKE 表之间的连接操作过滤掉没有选修任何课程的学生，只留下选修 CS112 课程并且只选修 CS112 课程的学生。这就是面向集合的思考和解决问题的方法。

B.3 At Most条件问题

At Most 条件[1] 问题是另一种我们可能经常遇到的查询问题。筛选出满足某个条件的行并不难，但有时候我们还需要限定符合条件的行的数目，这又该如何处理呢？下面的两个实例都与此类问题相关。

B.3.1 问题4

你希望找出最多选修两门课程的学生，没有选修任何课程的学生应该被排除在外。全体学生中，只有 AARON 选修的课程数目超过两门，他应该被排除在外。最终你希望得到如下所示的结果集。

```
SNO SNAME             AGE
--- ---------- ----------
  2 CHUCK              21
  3 DOUG               20
  4 MAGGIE             19
  5 STEVE              22
  6 JING               18
```

MySQL 和 PostgreSQL

使用聚合函数 COUNT 判断哪些学生最多选修了两门课程。

```
1 select s.sno,s.sname,s.age
2   from student s, take t
3  where s.sno = t.sno
4  group by s.sno,s.sname,s.age
5 having count(*) <= 2
```

DB2、Oracle 和 SQL Server

再次使用窗口函数 COUNT OVER 判断哪些学生最多选修了两门课程。

```
1  select distinct sno,sname,age
2    from (
3  select s.sno,s.sname,s.age,
4         count(*) over (
5           partition by s.sno,s.sname,s.age
6         ) as cnt
7    from student s, take t
 8   where s.sno = t.sno
```

注 1：意为“最多 XX 个”“不超过 XX 个”。——译者注

```
 9        )x
10   where cnt <= 2
```

讨论

以上两种解决方案的思路都是计算出 TAKE 表中每个 SNO 出现的次数。STUDENT 表和 TAKE 表的内连接操作能够确保剔除掉没有选修任何课程的学生。

原解决方案

The Essence of SQL 一书使用了聚合函数解决方案，如 MySQL 和 PostgreSQL 解决方案所示。同时，David Rozenshtein 也提供了另一个替代方案，该方案中使用了两次自连接，如下所示。

```
select distinct s.*
  from student s, take t
 where s.sno = t.sno
   and s.sno not in ( select t1.sno
                        from take t1, take t2, take t3
                       where t1.sno = t2.sno
                         and t2.sno = t3.sno
                         and t1.cno < t2.cno
                         and t2.cno < t3.cno )
```

上述这种两次自连接的解决方案很巧妙，因为它避免了聚合运算。下面来解释该做法的原理，先看一下子查询的 WHERE 子句。基于 SNO 的内连接操作能够确保子查询返回的每一行都是针对同一个学生的数据。那个比较 CNO 的条件用来判断一个学生选修的课程数量是否多于两门。子查询的 WHERE 子句可以这么表述："对于每一个学生，只返回第一个 CNO 小于第二个 CNO 并且第二个 CNO 小于第三个 CNO 的行。"如果一个学生选修的课程数量小于 3，那么该条件不会为真，因为没有第 3 个 CNO。也就是说，子查询就是为了找出选修了 3 门以上课程的学生。外层查询则负责返回至少选修了一门课程，并且 SNO 不存在于子查询返回结果的学生。

B.3.2　问题5

你希望找出年龄最多大于其他两名同学的学生。也就是说，本问题就是要找出那些比其他 0 个、1 个或者 2 个学生年龄大的学生。最终的结果集应该如下所示。

```
SNO SNAME      AGE
---- ---------- ---
   6 JING        18
   4 MAGGIE      19
   1 AARON       20
   9 GILLIAN     20
   8 KAY         20
   3 DOUG        20
```

MySQL 和 PostgreSQL

使用聚合函数 COUNT 和关联子查询找出比其他 0 个、1 个或 2 个学生年龄大的学生。

```
1 select s1.*
2   from student s1
```

```
3  where 2 >= ( select count(*)
4                from student s2
5               where s2.age <s1.age )
```

DB2、Oracle 和 SQL Server

使用窗口函数 DENSE_RANK 找出比其他 0 个、1 个或 2 个学生年龄大的学生。

```
1  select sno,sname,age
2    from (
3  select sno,sname,age,
4         dense_rank()over(order by age) as dr
5    from student
6         ) x
7   where dr <= 3
```

讨论

聚合函数解决方案使用标量子查询筛选出最多比其他两名学生年龄大的学生。我们把上述解决方案改写一下，添加一个标量子查询，这样你就容易理解其工作原理了。如下所示的实例中，CNT 列代表年龄比当前学生小的学生人数。

```
select s1.*,
       (select count(*) from student s2
         where s2.age < s1.age) as cnt
  from student s1
  order by 4

SNO SNAME             AGE        CNT
--- ---------- ---------- ----------
  6 JING               18          0
  4 MAGGIE             19          1
  1 AARON              20          2
  3 DOUG               20          2
  8 KAY                20          2
  9 GILLIAN            20          2
  2 CHUCK              21          6
  7 BRIAN              21          6
 10 CHAD               21          6
  5 STEVE              22          9
```

看过上面的查询结果，我们就能明白最终结果集是那些 CNT 值小于或等于 2 的学生。使用窗口函数 DENSE_RANK 的解决方案也类似于上面的相关子查询方案，它根据有多少人比当前学生年龄小计算出每个学生对应的排名（DENSE_RANK 不仅允许 Tie 的存在，还能保证名次连续，中间不留空白）。下面的查询展示了 DENSE_RANK 函数的输出结果。

```
select sno,sname,age,
       dense_rank()over(order by age) as dr
  from student

SNO SNAME             AGE         DR
--- ---------- ---------- ----------
  6 JING               18          1
  4 MAGGIE             19          2
  1 AARON              20          3
```

```
  3 DOUG              20          3
  8 KAY               20          3
  9 GILLIAN           20          3
  2 CHUCK             21          4
  7 BRIAN             21          4
 10 CHAD              21          4
  5 STEVE             22          5
```

最后把上述查询放入一个内嵌视图里，并只保留那些 DR 值小于或等于 3 的行。

原解决方案

The Essence of SQL 一书采取了一种有趣的解决办法，那就是重新表述问题。不去“找出最多比两位学生年龄大的学生”，而是“找出那些比其他 3 个或更多（至少 3 个）学生年龄小的学生。”如果你希望学习面向集合思考的问题解决方法，那么 *The Essence of SQL* 一书为我们做了一次非常精彩的示范，它强迫我们通过下面的两个步骤找到解决方案。

(1) 找出比其他 3 个或更多学生年龄大的学生集合。
(2) 返回不存在于上一步结果集的学生。

解决方案如下所示。

```
select *
  from student
 where sno not in (
select s1.sno
  from student s1,
       student s2,
       student s3,
       student s4
 where s1.age > s2.age
   and s2.age > s3.age
   and s3.age > s4.age
)

SNO SNAME      AGE
--- ---------- ---
  6 JING        18
  4 MAGGIE      19
  1 AARON       20
  9 GILLIAN     20
  8 KAY         20
  3 DOUG        20
```

如果自下而上仔细观察上述解决方案，我们会发现它先执行的是“找出比其他 3 个或更多学生年龄大的学生集合”，如下所示。（为提高可读性，使用 DISTINCT 压缩结果集。）

```
select distinct s1.*
  from student s1,
       student s2,
       student s3,
       student s4
 where s1.age > s2.age
   and s2.age > s3.age
   and s3.age > s4.age
```

```
SNO SNAME      AGE
--- ---------- ---
  2 CHUCK       21
  5 STEVE       22
  7 BRIAN       21
 10 CHAD        21
```

看到上述自连接查询，你可能感到迷惑不解，我们不妨先来分析一下 WHERE 子句。S1.AGE 比 S2.AGE 大，因此任何一个学生，只有他至少比一个学生年龄大，才可能会被保留下来。接下来，S2.AGE 比 S3.AGE 大，则年龄大于其他两个学生的学生会被保留下来。注意，大于具有传递性。如果 S1.AGE 比 S2.AGE 大，并且 S2.AGE 比 S3.AGE 大，那么 S1.AGE 当然也比 S3.AGE 大。一旦理解了这一点，我们就可以把上述查询简化到只有一个自连接，从而更便于我们理解每一步操作的输出结果。例如，找出至少比一个学生年龄大的学生。（除了年龄最小的 JING，其他所有学生都会被返回。）

```
select distinct s1.*
  from student s1,
       student s2
 where s1.age > s2.age

SNO SNAME      AGE
--- ---------- ---
  5 STEVE       22
  7 BRIAN       21
 10 CHAD        21
  2 CHUCK       21
  1 AARON       20
  3 DOUG        20
  9 GILLIAN     20
  8 KAY         20
  4 MAGGIE      19
```

接下来，找出比其他 2 个或更多学生年龄大的学生（JING 和 MAGGIE 会被排除掉）。

```
select distinct s1.*
  from student s1,
       student s2,
       student s3
 where s1.age > s2.age
   and s2.age > s3.age

SNO SNAME      AGE
--- ---------- ---
  1 AARON       20
  2 CHUCK       21
  3 DOUG        20
  5 STEVE       22
  7 BRIAN       21
  8 KAY         20
  9 GILLIAN     20
 10 CHAD        21
```

最后，找出比其他 3 个或更多学生年龄大的学生（只保留 CHUCK、STEVE、BRIAN 和 CHAD）。

```
select distinct s1.*
  from student s1,
       student s2,
       student s3,
       student s4
 where s1.age > s2.age
   and s2.age > s3.age
   and s3.age > s4.age

SNO SNAME      AGE
--- ---------- ---
  2 CHUCK       21
  5 STEVE       22
  7 BRIAN       21
 10 CHAD        21
```

现在我们知道了哪些学生比其他 3 个或更多学生年龄大，只需要在子查询中使用 NOT IN 就可以筛选出除了上述 4 人之外的那些学生。

B.4 At Least条件问题

At Most 条件的反面就是 At Least 条件[2]。处理 At Least 条件问题时，我们经常用到的技巧是把它转换为等价的 At Most 条件问题。我们可以把 At Least 条件重新表述为“不少于 XX 个”。

一般来说，如果我们能够从需求中识别出阈值，那么问题就算解决一半了。确立了阈值之后，我们可以选择解决问题的方式：从正面入手（使用聚合函数或窗口函数，例如 COUNT），或者从侧面入手（使用子查询和逻辑否定）。

B.4.1 问题6

你希望找出至少选修了两门课程的学生。

换一种方式表述问题的话可能有用，比如：“找出选修了两门以上课程的学生”，或者“找出选修的课程数量不少于两门的学生”。我们不妨使用前面的问题 4 中用过的技巧：使用聚合函数 COUNT 或窗口函数 COUNT OVER。最终的结果集应该如下所示。

```
SNO SNAME             AGE
--- ---------- ----------
  1 AARON              20
  3 DOUG               20
  4 MAGGIE             19
  6 JING               18
```

MySQL 和 PostgreSQL

使用聚合函数 COUNT 筛选出至少选修了两门课程的学生。

注 2：意为“最多 XX 个”。——译者注

```
1 select s.sno,s.sname,s.age
2   from student s, take t
3  where s.sno = t.sno
4  group by s.sno,s.sname,s.age
5 having count(*) >= 2
```

DB2、Oracle 和 SQL Server

使用窗口函数 COUNT OVER 筛选出至少选修了两门课程的学生。

```
 1  select distinct sno,sname,age
 2    from (
 3  select s.sno,s.sname,s.age,
 4         count(*) over (
 5           partition by s.sno,s.sname,s.age
 6         ) as cnt
 7    from student s, take t
 8   where s.sno = t.sno
 9         ) x
10   where cnt >= 2
```

讨论

问题 4 的“讨论”部分也适用于本问题的解决方案，它们使用的方法和技巧相同。聚合函数解决方案先把 STUDENT 表和 TAKE 表连接起来，然后在 HAVING 子句中使用 COUNT 筛选出那些选修了两门以上课程的学生。窗口函数解决方案则先连接 STUDENT 表和 TAKE 表，然后使用 STUDENT 表的全部列定义分区并执行 COUNT OVER 操作。最后，只要保留那些 CNT 大于或者等于 2 的行即可。

原解决方案

下面的解决方案使用 TAKE 表的自连接找出选修了两门以上课程的学生。子查询里的 SNO 相等条件能够确保每个学生只与自己的选课信息相比较。至于 CNO 大于比较条件，只有在一个学生至少选修了一门课程的情况下才会成立，否则 CNO 会等于另一个 CNO。(因为只有一门课程，只能和自身比较。）最后，返回子查询筛选出的那些学生，如下所示。

```
select *
  from student
 where sno in (
select t1.sno
 from take t1,
      take t2
 where t1.sno = t2.sno
   and t1.cno > t2.cno
)

SNO SNAME             AGE
--- ---------- ----------
  1 AARON              20
  3 DOUG               20
  4 MAGGIE             19
  6 JING               18
```

B.4.2　问题7

你希望找出同时选修了 CS112 和 CS114 两门课程的学生。这些学生可能也选修了其他课程，但他们必须同时选修 CS112 和 CS114。

本问题类似于问题 2，问题 2 要求学生只能选修其中一门课程，而本问题则要求学生两门课程都要选。（只有 AARON 和 DOUG 同时选修了 CS112 和 CS114。）我们只要修改问题 2 的解决方案，就能轻松解决本问题。最终的结果集应该如下所示。

```
SNO SNAME       AGE
--- ---------- ----
  1 AARON        20
  3 DOUG         20
```

MySQL 和 PostgreSQL

使用聚合函数 MIN 和 MAX 找出同时选修了 CS112 和 CS114 课程的学生。

```
1  select s.sno, s.sname, s.age
2    from student s, take t
3   where s.sno = t.sno
4     and t.cno in ('CS114','CS112')
5   group by s.sno, s.sname, s.age
6  having min(t.cno) != max(t.cno)
```

DB2、Oracle 和 SQL Server

使用窗口函数 MIN OVER 和 MAX OVER 找出同时选修了 CS112 和 CS114 课程的学生。

```
 1  select distinct sno, sname, age
 2    from (
 3  select s.sno, s.sname, s.age,
 4         min(cno) over (partition by s.sno) as min_cno,
 5         max(cno) over (partition by s.sno) as max_cno
 6    from student s, take t
 7   where s.sno = t.sno
 8     and t.cno in ('CS114','CS112')
 9         ) x
10   where min_cno != max_cno
```

讨论

以上两种解决方案的做法相同。IN 列表确保只有选修 CS112 或 CS114，或者同时两门都选了的学生才会被保留下来。如果一个学生没有同时选修这两门课程，那么 MIN(CNO) 就会等于 MAX(CNO)，进而该学生会被排除在外。为了直观地解释这一处理过程，我们不妨打印出窗口函数解决方案的中间结果。（为便于理解，这里增加了 T.CNO。）

```
select s.sno, s.sname, s.age, t.cno,
       min(cno) over (partition by s.sno) as min_cno,
       max(cno) over (partition by s.sno) as max_cno
  from student s, take t
 where s.sno = t.sno
   and t.cno in ('CS114','CS112')

SNO SNAME       AGE CNO   MIN_C MAX_C
```

```
--- ---------- ---- ----- ----- -----
  1 AARON        20 CS114 CS112 CS114
  1 AARON        20 CS112 CS112 CS114
  2 CHUCK        21 CS112 CS112 CS112
  3 DOUG         20 CS114 CS112 CS114
  3 DOUG         20 CS112 CS112 CS114
  4 MAGGIE       19 CS112 CS112 CS112
  6 JING         18 CS114 CS114 CS114
```

仔细观察上述结果集，我们会发现只有 AARON 和 DOUG 满足条件 `MIN(CNO) != MAX(CNO)`。

原解决方案

The Essence of SQL 一书中的解决方案用到了 `TAKE` 表的自连接，该解决方案如下所示。

```
select s.*
  from student s,
       take t1,
       take t2
 where s.sno  = t1.sno
   and t1.sno = t2.sno
   and t1.cno = 'CS112'
   and t2.cno = 'CS114'

SNO SNAME       AGE
--- ---------- ----
  1 AARON        20
  3 DOUG         20
```

以上所有解决方案都能确保不论最终结果集里的学生有没有选修其他课程，他们都同时选修了 `CS112` 和 `CS114`。如果你对上述自连接操作还不太理解，请试试看能否读懂下面的代码。

```
select s.*
  from take t1, student s
 where s.sno   = t1.sno
   and t1.cno  = 'CS114'
   and 'CS112' = any (select t2.cno
                        from take t2
                       where t1.sno  = t2.sno
                         and t2.cno != 'CS114')

SNO SNAME       AGE
--- ---------- ----
  1 AARON        20
  3 DOUG         20
```

B.4.3　问题8

找出那些至少比其他两位学生年龄大的学生。

把本问题重新表述一下可能更易于理解，“找出年龄大于两个以上同学的学生”。我们可以再次使用问题 5 的做法和技巧。最终的结果集如下所示。（只有 JING 和 MAGGIE 不符合条件。）

```
SNO SNAME           AGE
--- ---------- ----------
  1 AARON            20
  2 CHUCK            21
  3 DOUG             20
  5 STEVE            22
  7 BRIAN            21
  8 KAY              20
  9 GILLIAN          20
 10 CHAD             21
```

MySQL 和 PostgreSQL

使用聚合函数 COUNT 和关联子查询找出至少比其他两位学生年龄大的学生。

```
1 select s1.*
2   from student s1
3  where 2 <= ( select count(*)
4                 from student s2
5                where s2.age <s1.age )
```

DB2、Oracle 和 SQL Server

使用窗口函数 DENSE_RANK 找出至少比其他两位学生年龄大的学生。

```
1  select sno,sname,age
2    from (
3  select sno,sname,age,
4         dense_rank()over(order by age) as dr
5    from student
6         ) x
7   where dr >= 3
```

讨论

请参见问题 5 的“讨论”部分。上述两种解决方案的类似，唯一的差别在于最后是通过计数值还是排名来筛选出结果。

原解决方案

本问题由问题 6 衍生而来，不同之处在于这里只需要和 STUDENT 表打交道。适当修改问题 6 的解决方案就可以解决本问题，如下所示。

```
select distinct s1.*
  from student s1,
       student s2,
       student s3
 where s1.age > s2.age
   and s2.age > s3.age

SNO SNAME           AGE
--- ---------- ----------
  1 AARON            20
  2 CHUCK            21
  3 DOUG             20
  5 STEVE            22
  7 BRIAN            21
  8 KAY              20
  9 GILLIAN          20
 10 CHAD             21
```

B.5　Exactly问题

你可能会认为，“确认某些条件是否成立”并不难。许多时候也确实如此。不过，一旦牵扯到“精确的数量条件”（Exactly）时，这类问题也可能会瞬间变得棘手起来，尤其是当我们必须把多个表连接起来寻求答案时。问题的根源在于精确的数量条件具有排他性。有时候我们可以把“Exactly”理解成“Only”。不妨体会一下这两种表述的不同之处：“穿着鞋子的人”和“只穿着鞋子的人”。仅仅满足指定的条件（穿着鞋子）是不够的，我们还必须确保在满足该条件的同时，其他条件不会成立（确保没有穿鞋子之外的其他衣物）。

B.5.1　问题9

找出只讲授一门课程的教授。

教授们讲授哪些课程并不重要，我们关注的是他们是否只讲授一门课程。最终的结果集应该如下所示。

```
LNAME      DEPT           SALARY  AGE
---------- ---------- ---------- ----
POMEL      SCIENCE           500   65
```

MySQL 和 PostgreSQL

使用聚合函数 COUNT 找出只讲授一门课程的教授。

```
1  select p.lname,p.dept,p.salary,p.age
2    from professor p, teach t
3   where p.lname = t.lname
4   group by p.lname,p.dept,p.salary,p.age
5  having count(*) = 1
```

DB2、Oracle 和 SQL Server

使用窗口函数 COUNT OVER 找出只讲授一门课程的教授。

```
1  select lname, dept, salary, age
2    from (
3  select p.lname,p.dept,p.salary,p.age,
4         count(*) over (partition by p.lname) as cnt
5    from professor p, teach t
6   where p.lname = t.lname
7         ) x
8   where cnt = 1
```

讨论

通过内连接 PROFESSOR 表和 TEACH 表，我们能够确保把不讲授任何课程的教授都排除掉。聚合函数解决方案在 HAVING 子句里使用 COUNT 函数返回只讲授一门课程的教授。窗口函数解决方案使用 COUNT OVER 函数，但请留意 COUNT OVER 函数的 PARTITION 子句用到的 PROFESSOR 表的列，它们不同于聚合函数解决方案 GROUP BY 子句中的那些列。对于本例而言，GROUP BY 和 PARTITION BY 后面的列可以不相同，因为 PROFESSOR 表的 LNAME 列具有唯一性。也就是说，即使没有 P.DEPT、P.SALARY 和 P.AGE，COUNT 操作的结果也不会出错。在前几个实例中，我有意地在窗口函数解决方案的 PARTITION 子句和聚合函数解决方案的 GROUP BY 子句中放置

相同的列，这样做是为了告诉你 PARTITION 是动态的、更灵活的 GROUP BY。

原解决方案

The Essence of SQL 一书中的解决方案用到了和问题 3 同样的技巧：分两步找到答案。第一步是找到讲授了两门以上课程的教授。第二步是找到讲授了至少一门课程但不存在于第一步返回结果集的教授。完整的讨论参见问题 3，解决方案如下所示。

```
select p.*
  from professor p,
       teach t
 where p.lname = t.lname
   and p.lname not in (
select t1.lname
  from teach t1,
       teach t2
 where t1.lname = t2.lname
   and t1.cno   > t2.cno
)

LNAME      DEPT           SALARY        AGE
---------- ---------- ---------- ----------
POMEL      SCIENCE           500         65
```

B.5.2 问题10

你希望找出只选修 CS112 和 CS114 课程的学生（只选了这两门，没有选其他课程），但下面的查询返回的结果集为空。

```
select s.*
  from student s, take t
 where s.sno = t.sno
   and t.cno = 'CS112'
   and t.cno = 'CS114'
```

对于任何一行记录而言，一列不可能包含两个值（此处指的是简单数据类型，就像 STUDENT 表的值），因此上述查询不会筛选出任何有意义的结果。*The Essence of SQL* 一书中对于犯了类似错误的 SQL 语句已经做过详尽讨论。这里 DOUG 是唯一符合条件的学生，因此最终的查询结果应该只包含 DOUG。

MySQL 和 PostgreSQL

使用 CASE 表达式和聚合函数 COUNT 找出只选修 CS112 和 CS114 课程的学生。

```
1 select s.sno, s.sname, s.age
2   from student s, take t
3  where s.sno = t.sno
4  group by s.sno, s.sname, s.age
5 having count(*) = 2
6    and max(case when cno = 'CS112' then 1 else 0 end) +
7        max(case when cno = 'CS114' then 1 else 0 end) = 2
```

DB2、Oracle 和 SQL Server

使用窗口函数 COUNT OVER 和 CASE 表达式找出只选修 CS112 和 CS114 课程的学生。

```
 1  select sno,sname,age
 2    from (
 3  select s.sno,
 4         s.sname,
 5         s.age,
 6         count(*) over (partition by s.sno) as cnt,
 7         sum(case when t.cno in ( 'CS112', 'CS114' )
 8                  then 1 else 0
 9             end)
10         over (partition by s.sno) as both,
11         row_number()
12         over (partition by s.sno order by s.sno) as rn
13    from student s, take t
14   where s.sno = t.sno
15         ) x
16   where cnt  = 2
17     and both = 2
18     and rn   = 1
```

讨论

聚合函数解决方案的做法和问题 1 以及问题 2 的相同。STUDENT 表和 TAKE 表的内连接能够确保没有选修任何课程的学生被排除掉。HAVING 子句的 COUNT 表达式只保留选修两门课程的学生。CASE 表达式先计数课程数目，然后再求和。只有选修了 CS112 和 CS114 两门课程的学生，其 SUM 结果才可能等于 2。

窗口函数解决方案使用到的技巧也类似于问题 1 和问题 2。这里稍微不同的一点是，CASE 表达式的结果计算出来之后会被传递给窗口函数 SUM OVER。本解决方案另一个值得注意之处是，我们使用了窗口函数 ROW_NUMBER，从而避免使用 DISTINCT。不妨先去掉窗口函数解决方案最后的过滤条件，看一下中间结果，如下所示。

```
select s.sno,
       s.sname,
       s.age,
       count(*) over (partition by s.sno) as cnt,
       sum(case when t.cno in ( 'CS112', 'CS114' )
                then 1 else 0
           end)
       over (partition by s.sno) as both,
       row_number( )
       over (partition by s.sno order by s.sno) as rn
  from student s, take t
 where s.sno = t.sno

SNO SNAME   AGE  CNT BOTH   RN
--- ------ ---- ---- ---- ----
  1 AARON    20    3    2    1
  1 AARON    20    3    2    2
  1 AARON    20    3    2    3
  2 CHUCK    21    1    1    1
  3 DOUG     20    2    2    1
  3 DOUG     20    2    2    2
  4 MAGGIE   19    2    1    1
  4 MAGGIE   19    2    1    2
```

```
  5 STEVE     22    1    0    1
  6 JING      18    2    1    1
  6 JING      18    2    1    2
```

仔细观察上述结果，我们看到最终的结果集是 BOTH 和 CNT 都等于 2 的那一行。其实 RN 值等于 1 或 2 都没有关系，该列的作用在于去掉重复项，因此我们无须使用 DISTINCT。

原解决方案

The Essence of SQL 一书中的解决方案使用含有多个自连接的子查询首先找到至少选修了 3 门课程的学生。然后，使用 TAKE 表的自连接找出选修 CS112 和 CS114 的学生。最后筛选出选修 CS112 和 CS114，但选修课程数量又不多于两门的学生。解决方案如下所示。

```
select s1.*
  from student s1,
       take t1,
       take t2
 where s1.sno  = t1.sno
   and s1.sno  = t2.sno
   and t1.cno = 'CS112'
   and t2.cno = 'CS114'
   and s1.sno not in (
select s2.sno
  from student s2,
       take t3,
       take t4,
       take t5
 where s2.sno = t3.sno
   and s2.sno = t4.sno
   and s2.sno = t5.sno
   and t3.cno > t4.cno
   and t4.cno > t5.cno
)

SNO SNAME      AGE
--- ---------- ---
  3 DOUG        20
```

B.5.3　问题11

你希望找出比其他两位学生年龄大的学生。本问题的另一种陈述方式是：找出按照年龄从小到大排序排在第三位的学生。最终的结果集应该如下所示。

```
SNO SNAME             AGE
--- ---------- ----------
  1 AARON              20
  3 DOUG               20
  8 KAY                20
  9 GILLIAN            20
```

MySQL 和 PostgreSQL

使用聚合函数 COUNT 和关联子查询找出按照年龄从小到大排序排在第三位的学生。

```
1 select s1.*
2   from student s1
3  where 2 = ( select count(*)
4                from student s2
5               where s2.age <s1.age )
```

DB2、Oracle 和 SQL Server

使用窗口函数 DENSE_RANK 找出按照年龄从小到大排序排在第三位的学生。

```
1  select sno,sname,age
2    from (
3  select sno,sname,age,
4         dense_rank()over(order by age) as dr
5    from student
6         ) x
7   where dr = 3
```

讨论

聚合函数解决方案使用标量子查询找出年龄大于其他两位（且只有两位）同学的学生。为了更清楚地展示处理过程，下面我们改写上述解决方案的代码。在下面的例子中，CNT 列代表比当前学生年龄小的学生人数。

```
select s1.*,
       (select count(*) from student s2
         where s2.age < s1.age) as cnt
  from student s1
  order by 4

SNO SNAME             AGE        CNT
--- ---------- ---------- ----------
  6 JING               18          0
  4 MAGGIE             19          1
  1 AARON              20          2
  3 DOUG               20          2
  8 KAY                20          2
  9 GILLIAN            20          2
  2 CHUCK              21          6
  7 BRIAN              21          6
 10 CHAD               21          6
  5 STEVE              22          9
```

改写后的查询便于我们看清楚谁是按照年龄从小到大排序排在第三位的学生（也就是 CNT 值等于 2 的学生）。

使用了窗口函数 DENSE_RANK 的解决方案类似于上述标量子查询的做法，根据有多少位学生比当前学生年龄小为每一个学生排名。（DENSE_RANK 不仅允许 Tie 的存在，也能保证名次连续，中间不留空白。）下面的查询展示了 DENSE_RANK 函数的输出结果。

```
select sno,sname,age,
       dense_rank()over(order by age) as dr
  from student
```

```
SNO SNAME             AGE         DR
--- ---------- ---------- ----------
  6 JING               18          1
  4 MAGGIE             19          2
  1 AARON              20          3
  3 DOUG               20          3
  8 KAY                20          3
  9 GILLIAN            20          3
  2 CHUCK              21          4
  7 BRIAN              21          4
 10 CHAD               21          4
  5 STEVE              22          5
```

最后把上述查询放入内嵌视图中，并只保留 `DR` 值等于 3 的行。

原解决方案

The Essence of SQL 一书中的解决方案分为两步：首先找出比 3 名以上学生年龄大的学生。然后找出比两位以上学生年龄大的学生，但又不属于第一步返回结果集的学生。David Rozenshtein 这样表述其思路："找出至少比两名学生年龄大，但又不比 3 名以上学生年龄大的学生。"解决方案如下所示。

```
select s5.*
  from student s5,
       student s6,
       student s7
 where s5.age > s6.age
   and s6.age > s7.age
   and s5.sno not in (
select s1.sno
  from student s1,
       student s2,
       student s3,
       student s4
 where s1.age > s2.age
   and s2.age > s3.age
   and s3.age > s4.age
)

SNO SNAME      AGE
--- ---------- ---
  1 AARON       20
  3 DOUG        20
  9 GILLIAN     20
  8 KAY         20
```

上述解决方案做法与问题 5 相同。你不妨参考问题 5 的"讨论"部分，体会一下如何充分利用自连接查询解决问题。

B.6　Any和All问题

涉及"Any"（任意一个）或"All"（全部）的查询通常要求我们找出完全符合一个条件或多个条件的行。例如，如果要找出"吃所有蔬菜的人"，实际上要寻找的人是"没有任

何一种蔬菜他们不吃”。这种问题通常被归类为“关系除法”（relational division）。对于“Any”问题，很重要的一点是，我们要特别注意问题的陈述方式。考虑如下两种不同的问题陈述：“选修了任意一门课程的学生”和“比任何火车都快的飞机”。前者要求的是“找出选修了至少一门课程的学生”，而后者的要求是“找出比所有火车都快的飞机”。

B.6.1　问题12

你希望找出选修了全部课程的学生。

在 TAKE 表中一个学生选修的课程总数必须等于 COURSES 表中所有课程的总数。COURSES 表里有 3 门课程。只有 AARON 选修了全部 3 门课程，因此 AARON 应该是唯一被返回的学生。最终的结果集应该如下所示。

```
SNO SNAME   AGE
--- ------ ----
  1 AARON    20
```

MySQL 和 PostgreSQL

使用聚合函数 COUNT 找出选修所有课程的学生。

```
1 select s.sno,s.sname,s.age
2   from student s, take t
3  where s.sno = t.sno
4  group by s.sno,s.sname,s.age
5 having count(t.cno) = (select count(*) from courses)
```

DB2 和 SQL Server

使用窗口函数 COUNT OVER，并使用外连接而不是子查询。

```
 1  select sno,sname,age
 2    from (
 3  select s.sno,s.sname,s.age,
 4         count(t.cno)
 5         over (partition by s.sno) as cnt,
 6         count(distinct c.title) over() as total,
 7         row_number() over
 8         (partition by s.sno order by c.cno) as rn
 9    from courses c
10         left join take t    on (c.cno = t.cno)
11         left join student s on (t.sno = s.sno)
12         ) x
13   where cnt = total
14     and rn = 1
```

Oracle

上述 DB2 解决方案也适用于 Oracle 9*i* 及后续版本。除此之外，也可以使用 Oracle 专有的外连接句语法。对于 Oracle 8*i* 及更早版本而言，则只能使用该解决方案。

```
 1  select sno,sname,age
 2    from (
 3  select s.sno,s.sname,s.age,
 4         count(t.cno)
```

```
 5         over (partition by s.sno) as cnt,
 6         count(distinct c.title) over() as total,
 7         row_number() over
 8         (partition by s.sno order by c.cno) as rn
 9    from courses c, take t, student s
10   where c.cno = t.cno (+)
11     and t.sno = s.sno (+)
12         )
13   where cnt = total
14     and rn  = 1
```

讨论

聚合函数解决方案使用子查询返回课程总数，外层查询负责筛选出选修课程数量等于子查询返回值的学生。窗口函数解决方案采取了另一种做法：它外连接到 COURSES 表，而不是子查询。窗口函数解决方案使用窗口函数返回一个学生选修的课程数量（别名 CNT）和 COURSES 表里的课程总数（别名 TOTAL）。下面的查询展示了这些窗口函数的中间结果。

```
select s.sno,s.sname,s.age,
       count(distinct t.cno)
       over (partition by s.sno) as cnt,
       count(distinct c.title) over() as total,
       row_number( )
       over(partition by s.sno order by c.cno) as rn
  from courses c
       left join take t    on (c.cno = t.cno)
       left join student s on (t.sno = s.sno)
 order by 1

SNO SNAME   AGE  CNT      TOTAL   RN
--- ------ ---- ---- ---------- ----
  1 AARON    20    3          3    1
  1 AARON    20    3          3    2
  1 AARON    20    3          3    3
  2 CHUCK    21    1          3    1
  3 DOUG     20    2          3    1
  3 DOUG     20    2          3    2
  4 MAGGIE   19    2          3    1
  4 MAGGIE   19    2          3    2
  5 STEVE    22    1          3    1
  6 JING     18    2          3    1
  6 JING     18    2          3    2
```

CNT 和 TOTAL 值相等的行就是选修全部课程的学生。这里没有使用 DISTINCT，而是使用 ROW_NUMBER 剔除掉重复项。严格地说，TAKE 表和 STUDENT 表的外连接并不是必须的，因为每一门课程都被选修过。如果有一门课程大家都没有选，则 CNT 和 TOTAL 不可能相等，并且对应行的 SNO、SNAME 和 AGE 都是 Null。在下面的例子中，我们添加了一门新课程，这门课程当然不曾有学生选过。通过下面的查询我们可以看到，如果存在一门所有学生都没有选的课程会出现什么样的中间结果。（为了便于理解，查询结果增加了 C.TITLE 列。）

```
insert into courses values ('CS115','BIOLOGY',4)

select s.sno,s.sname,s.age,c.title,
```

```
        count(distinct t.cno)
        over (partition by s.sno) as cnt,
        count(distinct c.title) over() as total,
        row_number( )
        over(partition by s.sno order by c.cno) as rn
   from courses c
        left join take t    on (c.cno = t.cno)
        left join student s on (t.sno = s.sno)
  order by 1

SNO SNAME  AGE TITLE      CNT TOTAL  RN
--- ------ --- ---------- --- ----- ---
  1 AARON   20 PHYSICS      3     4   1
  1 AARON   20 CALCULUS     3     4   2
  1 AARON   20 HISTORY      3     4   3
  2 CHUCK   21 PHYSICS      1     4   1
  3 DOUG    20 PHYSICS      2     4   1
  3 DOUG    20 HISTORY      2     4   2
  4 MAGGIE  19 PHYSICS      2     4   1
  4 MAGGIE  19 CALCULUS     2     4   2
  5 STEVE   22 CALCULUS     1     4   1
  6 JING    18 CALCULUS     2     4   1
  6 JING    18 HISTORY      2     4   2
               BIOLOGY      0     4   1
```

仔细观察上述结果，很容易就能看出任何一行都无法满足最后的那个过滤条件。另外还要注意的一点是，窗口函数会在 WHERE 子句之后执行，因此计算 COURSES 表的课程总数时，需要使用 DISTINCT。（否则，计算出来的结果就是所有学生选课的总人次，即 select count(cno) from take。）



上述 TAKE 表中不存在重复项，因此该解决方案能正常执行。如果 TAKE 表有重复项，例如一个学生 3 次选修了同一门课程，那么该解决方案就不对了。简单改动上述解决方案就可以解决重复项问题：COUNT(T.CNO) 时加上 DISTINCT。

原解决方案

The Essence of SQL 一书中的解决方案没有使用聚合函数，却用了笛卡儿积，其做法非常巧妙。下面的查询就是该解决方案。

```
select *
  from student
 where sno not in
      ( select s.sno
          from student s, courses c
         where (s.sno,c.cno) not in (select sno,cno from take) )
```

David Rozenshtein 重新表述了该问题：“针对每一个学生，找出他们没有选过的课程，最后如果有谁不在其中，则他必定选修了全部课程。”以这种方式看待本问题的话，就回到了前面讨论过的逻辑否定问题。重新回顾一下 David Rozenshtein 关于处理逻辑否定问题的建议：“解决逻辑否定问题有两个步骤，首先要想找出‘谁没有做某事’，就先找到‘谁做了某事’，然后排除掉这些人。”

最内层的子查询会返回所有有效的 SNO/CNO 组合。中间一层的子查询使用 STUDENT 表和 COURSES 表之间的笛卡儿积，返回（假设每一个学生都选修了全部课程）所有可能的 SNO/CNO 组合，进而过滤掉有效的 SNO/CNO 组合（只留下了实际上不存在的 SNO/CNO 组合）。对于最外层的查询，只有当 SNO 不存在于中间一层子查询结果时，才会被保留下来。看一下如下所示的几个查询，你可能就会更清楚本解决方案的思路。为提高可读性，这里只用到了 AARON 和 CHUCK（只有 AARON 选修了所有课程）。

```
select *
  from student
 where sno in ( 1,2 )

SNO SNAME       AGE
--- ---------- ----
  1 AARON        20
  2 CHUCK        21

select *
  from take
 where sno in ( 1,2 )

SNO CNO
--- -----
  1 CS112
  1 CS113
  1 CS114
  2 CS112

select s.sno, c.cno
  from student s, courses c
 where s.sno in ( 1,2 )
 order by 1

SNO CNO
--- -----
  1 CS112
  1 CS113
  1 CS114
  2 CS112
  2 CS113
  2 CS114
```

上述查询分别打印出了 STUDENT 表中与 AARON 和 CHUCK 相关的行，AARON 和 CHUCK 选修过的课程，以及假设 AARON 和 CHUCK 选修了全部课程情况下的笛卡儿积。笛卡儿积中与 AARON 相关的行和 TAKE 表中的数据相同，但笛卡儿积中与 CHUCK 相关的数据，其中有两行实际上不存在于 TAKE 表。下面给出了中间一层的子查询，它使用 NOT IN 排除掉有效的 SNO/CNO 组合。

```
select s.sno, c.cno
  from student s, courses c
 where s.sno in ( 1,2 )
   and (s.sno,c.cno) not in (select sno,cno from take)
```

```
SNO CNO
--- -----
  2 CS113
  2 CS114
```

注意，AARON 没有出现在中间一层子查询的返回值里（因为 AARON 选修了全部课程）。中间一层子查询的结果集代表笛卡儿积中 CHUCK 没有选的那些课程。最后执行最外层的查询，如果 SNO 不存在于中间一层子查询结果集，则 STUDENT 表中相应的行会被保留下来。

```
select *
  from student
 where sno in ( 1,2 )
   and sno not in
      ( select s.sno from student s, courses c
         where s.sno in ( 1,2 )
           and (s.sno,c.cno) not in (select sno,cno from take))

SNO SNAME       AGE
--- ---------- ----
  1 AARON        20
```

B.6.2　问题13

找出比任何其他学生年龄都大的学生。

我们可以把本问题重新表述为："找出年龄最大的学生。"最终的结果集应该如下所示。

```
SNO SNAME       AGE
--- ---------- ----
  5 STEVE        22
```

MySQL 和 PostgreSQL

在子查询中使用聚合函数 MAX 找出年龄最大的学生。

```
1 select *
2   from student
3  where age = (select max(age) from student)
```

DB2、Oracle 和 SQL Server

在内嵌视图中使用窗口函数 MAX OVER 找出年龄最大的学生。

```
1  select sno,sname,age
2    from (
3  select s.*,
4         max(s.age)over() as oldest
5    from student s
6         ) x
7   where age = oldest
```

讨论

以上两种解决方案都使用了 MAX 函数找出年龄最大的学生。子查询解决方案首先计算出 STUDENT 表中最大的年龄，然后把计算结果传递给外层查询，而外层查询据此找出年龄等于该计算值的学生。窗口函数解决方案的做法和子查询解决方案一样，但它为每一行都返

回了最大的年龄值。窗口函数解决方案的中间结果如下所示。

```
select s.*,
       max(s.age) over() as oldest
  from student s

SNO SNAME       AGE     OLDEST
--- ---------- ---- ----------
  1 AARON        20         22
  2 CHUCK        21         22
  3 DOUG         20         22
  4 MAGGIE       19         22
  5 STEVE        22         22
  6 JING         18         22
  7 BRIAN        21         22
  8 KAY          20         22
  9 GILLIAN      20         22
 10 CHAD         21         22
```

要找到年龄最大的学生，只要保留 AGE=OLDEST 的行即可。

原解决方案

The Essence of SQL 一书中的解决方案在子查询中使用 STUDENT 表自连接，以找出所有年龄小于其他学生的学生。外层查询则负责筛选出 STUDENT 表中不存在于上述子查询返回结果集中的学生。这一处理过程可以理解为“先找出至少比其他一位学生年龄小的学生，其他不存在于该查询结果集内的学生就是答案”。

```
select *
  from student
 where age not in (select a.age
                     from student a, student b
                    where a.age < b.age)
```

上述子查询使用笛卡儿积找出 A 表中年龄小于 B 表的学生的年龄值。唯一不会比其他年龄值小的就是最大年龄值。上述子查询不会返回该最大值。外层查询使用 NOTIN，从 STUDENT 表中筛选出所有 AGE 不存在于上述子查询返回结果集的行。（在子查询中，如果 A.AGE 被返回，这就意味着 STUDENT 表中存在一个更大的 AGE 值。）如果你不理解这个处理过程，不妨仔细看一下如下所示的查询语句。从概念上来讲，它们的原理相似，但下面的查询可能更具一般性。

```
select *
  from student
 where age >= all (select age from student)
```

作者简介

Anthony Molinaro 是 Wireless Generation 公司的数据库开发人员，在帮助开发人员改善 SQL 查询效率方面拥有多年的丰富经验。Anthony 专精 SQL 技术，他的客户都知道遇到棘手的 SQL 问题时找他帮忙就对了。他博览群书，对关系理论有深入研究，并拥有长达九年的一线 SQL 开发经验。Anthony 十分熟悉那些新近加入标准的 SQL 新特性，例如窗口函数。

封面介绍

我们的图书封面之所以与众不同，是因为我们不断倾听读者的批评和建议，我们自己也在反复实验，再加上来自发行渠道的反馈意见，多重作用之下我们得以形成自己的特色。与众不同的封面凸显出我们对技术主题的独特解读方式，赋予了原本枯燥的内容以鲜活的生命力和个性。

本书的封面动物是鬣蜥（Agamid lizard）。这些蜥蜴属于鬣蜥科，物种多达 300 多个。鬣蜥广泛见于非洲、亚洲、澳大利亚和南欧，普遍具有强壮的腿部，某些变种有变色能力。不像其他的蜥蜴，鬣蜥的尾巴断掉后不会重新长出来。它们能存活于多种环境下，从干旱的沙漠到温暖潮湿的热带雨林都能找见它们的踪影。

部分品种作为宠物十分受大家欢迎。这其中就有鬃狮蜥（Bearded Dragon，学名 Pogona）。它们性情温和，好奇心强，身体最长可达大约 50 厘米。尽管身材纤巧，它们仍然被称作“巨型”蜥蜴，也需要宽敞的生活空间。雄性的鬃狮蜥通常具有领地意识。虽然它们具有社会性，但是过度拥挤的环境会令它们感到紧张，尤其是当它们无处藏身时。过度拥挤也会引发不同个体之间的争斗以致受伤，争斗中的个体可能会失去脚趾和尾巴，并导致食欲不振。

鬃狮蜥的头部呈三角形，下巴以上有许多突出的棘状鳞。这些棘状鳞看起来像络腮胡子（因此而得名）。在其身体侧面也会有棘状鳞。鬃狮蜥会张开嘴展示它们尖锐的颚须以吓阻敌人和同类。它们也会展平身体，这时看起来显得更大。作为宠物，当它们面对主人并感觉舒适安全的时候，也可能会收起突起的棘状鳞。

尽管鬃狮蜥源自澳大利亚，但美国商家出售的都是欧洲进口的鬃狮蜥的后代。因为澳大利亚法律严格限制野生动物的出口。

飞蜥（dracovolans）是另一个变种。体长不超过 27 厘米，身体颀长，肋部有翼膜。雄性往往会占据两到三棵树作为领地，每棵树上可能住有一只到三只雌性。为了能从此处迅速移动到彼处，它会从树上或者其他高处展开翼膜滑翔。然而，它们通常无法在雨中或者风中飞行。受到威胁时，飞蜥也会展开翼膜，让自己看起来更大一些。

另一个有趣的变种是彩虹飞蜥（Agama agama），见于撒哈拉以南非洲。这种生物常常群居，每个族群 10 只到 20 只不等，族群领袖通常为年长的雄性。晚间它们的身体呈现黑褐色，但是黎明时转为淡蓝色，头部和尾部呈现出鲜艳的橘色。皮肤颜色也会根据情绪发生变化，就像一只虚拟的情绪戒指。例如，雄性飞舞时，它们的头部将变为棕色，身体上则出现白色的斑点。

本书的责任编辑是 Darren Kelly，Kenneth Kimball 是文字编辑，Karmyn Guthrie 负责校对。nSight 公司提供了产品服务。Jamie Peppard 和 Genevieve d'Entremont 负责品质管理。Jansen Fernald 提供了产品支持。Beth Palmer 完成了索引部分。

Karen Montgomery 基于 Edie Freedman 的一系列设计作品设计出了本书的封面。封面的图像来自 Dover Pictorial Archive 的一幅 19 世纪雕刻作品。Karen Montgomery 借助 Adobe InDesign CS 创作了封面布局，并使用到了 Adobe 的 ITC Garamond 字体。

David Futato 设计了内页布局。Keith Fahlgren 使用一个格式转换工具把本书的内容转换至 FrameMaker 5.5.6，该格式转换工具由 Erik Ray、Jason McIntosh、Neil Walls 和 Mike Sierra 合作完成，使用到了 Perl 和 XML 技术。文本字体是 Linotype Birka，标题字体是 Adobe Myriad Condensed，代码字体是 LucasFont 的 Sans Mono Condensed。书中的插图由 Robert Romano、Jessamyn Read 和 Lesley Borash 合作完成，他们用到了 Macromedia FreeHand MX 和 Adobe Photoshop CS。Christopher Bing 绘制了提示和警告图标。Jansen Fernald 完成了本书末尾的版权声明页。

O'REILLY®

SQL经典实例

了解SQL查询语言的基本原理，但仍感觉无法自由运用SQL？想在数据上线前用SQL跑一遍数据库？想进一步提高SQL技能？

以上需求，《SQL经典实例》都可以满足。本书致力于让广大数据库相关人员了解SQL的强大数据处理能力。书中汇集了150多个SQL示例，给出了常见问题的解决方案，帮助读者扩宽思路并用学到的技术来解决新问题，既适合SQL初学者更深入理解SQL，也适合SQL中高级用户进阶及日常查询参考。

- where子句等常见查询语句
- 查询结果排序
- 连接查询
- 如何获取数据库元信息
- 常见数字运算实例
- 字符串处理
- 日期处理
- 数据仓储和复杂报表生成领域的查询
- 与层次化有关的一些实例

“一旦你发现SQL能帮你大大减少工作量，你就再也离不开这本书了。”

——**Olivier Pomel**
Wireless Generation公司研发总监

安东尼·莫利纳罗（Anthony Molinaro），专精SQL技术，擅长解决棘手的SQL问题，对关系理论有深入研究。

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600



分类建议 计算机 / 数据库 / SQL

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)



ISBN 978-7-115-48420-8
定价：149.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks